Apache Lucene

Sistemas de busca com técnicas de Recuperação de Informação

Casa do Código

MARCO REIS

Sumário

ISBN

Impresso e PDF: 978-85-7254-027-8

EPUB: 978-85-7254-028-5

MOBI: 978-85-7254-029-2

Caso você deseje submeter alguma errata ou sugestão, acesse http://erratas.casadocodigo.com.br.

Agradecimentos

Escrever um livro não é fácil. São muitas horas de trabalho, muitas noites, dias e fins de semana escrevendo, planejando, testando, refazendo, atualizando... O trabalho não tem fim. Por isso agradeço à minha família pela compreensão em tantos dias (e noites) de ausência. Mesmo quando estava presente, muitas vezes minha mente voava em linhas de código, capítulos e parágrafos a escrever.

Como autor, tenho que agradecer à pessoa para quem esse livro foi escrito: o leitor. O objetivo sempre foi passar esse conhecimento para você. Espero que ajude e incentive muitas discussões sobre o assunto neste momento em que a tecnologia avança com tamanha velocidade.

Prefácio

A internet mudou a forma como vivemos e hoje utilizamos as ferramentas de busca para tudo no nosso cotidiano. Em grande parte, esse sucesso se deve a ferramentas como o Google, Yahoo! e o Bing da Microsoft.

Não é necessário ir à biblioteca fazer pesquisas, ler um livro ou olhar uma enciclopédia, porque esses recursos estão disponíveis na internet. Se você quiser saber sobre um assunto é só pesquisá-lo em um buscador. Existe uma urgência por informação, inclusive com implicações financeiras. A informação tem uma posição central na sociedade moderna.

Mas o usuário não busca apenas texto. Ele também busca imagem, som, vídeo, mapas e recomendações. Até mesmo as pesquisas textuais sofreram mudanças. Ela deve incluir sinônimos, correção ortográfica e até tradução. Além da busca geral em toda a internet, temos buscadores especializados em artigos acadêmicos, empregos, redes sociais, imagens e até ciências. Existem muitas possibilidades para criar soluções interessantes para o usuário, que vão além da simples busca por palavra-chave.

A seguir vamos listar outros buscadores relevantes que temos atualmente. Ok, o mais usado certamente é o Google, mas temos algumas surpresas. Veja:

- Google Scholar: busca em artigos científicos;
- YouTube: um dos buscadores mais usados;
- DuckDuckGo: ótima alternativa para navegação anônima;
- Wolfram Alpha: utiliza mecanismos de inteligência artificial para responder perguntas;
- Twitter: análise em tempo real das tendências;
- Facebook: forte concorrente do Google no mercado de buscas e vídeos;
- LinkedIn: além da rede social, tem um mecanismo de busca por perfis profissionais.

E temos muitos buscadores por um motivo. Neste último século houve um grande aumento na quantidade de informação disponível e o ser humano tem o desejo natural de compartilhar esse conhecimento. A internet acabou sendo o meio mais eficiente para armazenar esse conteúdo. Mas, sem uma boa ferramenta de busca, como iríamos encontrar o que queremos ou precisamos? Mais do que isso, para que serve a informação que não pode ser encontrada?

A mesma situação acontece nas empresas. Elas estão criando cada vez mais informação. Um exemplo simples é o log dos sistemas web. Verifique a quantidade diária de log gerado pelas aplicações. Mas esse tipo de informação só é armazenado porque existe um motivo que justifique o custo. Não é apenas para encontrar erros do sistema. Podemos utilizar para atividades mais interessantes como entender o comportamento do usuário. Em todos os casos, precisamos de uma boa ferramenta de análise que encontre exatamente a informação que procuramos entre milhões (muitas vezes bilhões) de linhas. E isso deve acontecer em milésimos de segundo.

Ferramentas com grande volume de dados como Twitter, LinkedIn e Evernote têm equipes específicas para desenvolvimento de novas funcionalidade de busca para melhorar a qualidade de seu serviço. São casos em que lidamos com milhões de usuários, com busca em tempo real, sistemas de recomendação e autocompletar.

Meu interesse pela área surgiu de uma necessidade parecida. Eu trabalho em um tribunal e a quantidade de informação gerada é incrível. São centenas de milhares de documentos sendo criados diariamente. Tudo isso é conhecimento. São profissionais especializados trabalhando em uma área crítica da sociedade. Percebi que existia uma dificuldade grande para encontrar os documentos depois de algum tempo. Em função dessa demanda comecei a pesquisar soluções de busca e encontrei várias ferramentas interessantes. Este livro vai tratar de uma delas, que é o Apache Lucene. Mas existem ferramentas auxiliares ou

complementares como o Nutch, que será estudado neste livro, e outras que não veremos, como o Solr, o OpenNLP, o Hadoop e o Cassandra. Talvez em um próximo livro.

Falaremos da necessidade de informação do usuário moderno, que está acostumado a encontrar o que procura em ferramentas como o Google ou Bing, com todas as suas facilidades. O usuário quer a mesma funcionalidade de busca nos seus sistemas. Windows ou Mac OS, sistema de RH, ERP, CRM, no sistema da locadora ou qualquer aplicação que você esteja desenvolvendo nesse momento. Mesmo que seu usuário não esteja esperando, você pode disponibilizar uma boa ferramenta de busca e surpreendê-lo.

Quem deve ler este livro

Este é um livro técnico, com código-fonte e conceitos intermediários sobre programação. O ideal é que o leitor tenha conhecimento em Java. Não é necessária muita experiência, mas seria bem proveitoso se você se sentisse confortável com conceitos como Orientação a Objetos e Lógica de Programação. Também não é essencial, mas saber um pouco sobre banco de dados relacional e normalização de dados pode ajudar, porque alguns exemplos comparam o Lucene com um banco de dados, mostrando as vantagens e desvantagens de cada um. Em algumas seções falamos sobre JPA e JSF, apenas para gravar os dados simulando uma aplicação real.

O conteúdo é extenso e pode ser aproveitado por diversos perfis profissionais:

- Programadores: em última instância, são os responsáveis pela implementação do código disponível neste livro;
- Gerentes de projeto: podem utilizar este conteúdo para enriquecer funcionalidades já existentes nos projetos ou até

mesmo propor novas aplicações diferentes e mais avançadas que os sistemas tradicionais;
- Arquitetos de software: são estudados muitos casos de integração e arquitetura de motores de busca. A equipe de arquitetura pode encontrar soluções interessantes para os problemas do cotidiano;
- DBAs: as ferramentas e soluções estudadas no livro podem ser complementares aos bancos de dados tradicionais e são úteis em tarefas que exigem processamento intensivo de dados.

Organização dos capítulos

O livro está dividido em 2 partes. A primeira parte mostra como criar motores de busca com o Lucene, e a segunda mostra técnicas avançadas da área, incluindo o uso de outras ferramentas, como o Hibernate Search e o Apache Nutch. No total são 10 capítulos, cada um tratando de um assunto específico, sendo que no final de cada um temos uma lista de referências para completar os estudos.

O primeiro capítulo, *Introdução*, mostra o que cada ferramenta é capaz de fazer, bem como suas características. No final deste capítulo estão listados os repositórios de código-fonte do livro, disponíveis no GitHub.

O capítulo *2. Conceitos de recuperação da informação* fala da teoria aplicada aos sistemas de busca. Inclui ideias básicas como índice, documento, coleção de documentos e metadados. Mas também explicamos tópicos mais avançadas como o índice invertido, TF-IDF, processamento de linguagem natural e *text mining*. Nesse caso, não estamos falando especificamente de uma tecnologia ou biblioteca, mas sim da teoria envolvida no tema.

O capítulo *3. Indexação e busca* mostra exemplos de como indexar e buscar informações nos dados do seu computador ou da sua

empresa. É um exemplo simples, onde vamos ver as principais classes e funcionalidades necessárias para criar nosso primeiro motor de busca.

O capítulo *4. Tipos de busca* fala sobre as opções disponíveis no Lucene, não apenas por palavra-chave, mas também busca por intervalo de valores, conectivos e busca por valor aproximado. Nesse ponto é possível imaginar como substituir algumas funcionalidades existentes nos seus sistemas pelo Lucene.

Estudaremos o núcleo do Lucene em *5. Principais classes do Lucene*. No capítulo *6. Configurações avançadas* vemos algumas otimizações que podem ser feitas para indexação e busca. É onde serão vistos os principais parâmetros e configurações que alteram o comportamento e performance do Lucene.

No capítulo *7. Integração com sistemas corporativos* temos um modelo de integração entre uma aplicação Java e o Lucene. Para sistemas que usam JPA e Hibernate, há o capítulo 8 sobre o *Hibernate Search ORM*. Foram criadas duas aplicações para mostrar essas possibilidades. Não há uma única forma correta de se integrar essas ferramentas, então são mostradas algumas possibilidades com Lucene e com Hibernate Search.

O capítulo *9. Recursos avançados* apresenta técnicas como *highlight*, criação de sinônimos, sugestão de termos, classificação de documentos, otimizações, análise linguística, *text mining* e testes de performance. São complementos que estão além da busca por palavra-chave.

No capítulo *10. Extraindo dados da internet* veremos técnicas e ferramentas para *web crawling* e *web scraping*. Assim, criamos um robô para navegar na internet e extrair o conteúdo das páginas visitadas. Depois de baixar esse conteúdo, procedemos com a indexação, tornando esses textos disponíveis para consulta local na nossa aplicação.

Por fim, o capítulo *11. Referências do livro* mostra todos os recursos usados nos capítulos anteriores.

Sobre o autor

Marco Reis é Engenheiro de Software e trabalha em projetos de *big data* e inteligência artificial. Mestre em Computação Aplicada pela Universidade de Brasília, tem artigos publicados e apresentados em congressos nacionais e internacionais nas áreas de *Big Data* e Computação em Nuvem. Como desenvolvedor de software tem experiência em Java, *microservices*, *messaging* e NLP. Seu site pessoal é http://marcoreis.net., no qual escreve artigos sobre tecnologia.

Capítulo 1
Introdução

Estamos na *Era Digital*, em que há muito conhecimento disponível tanto na internet quanto nos sistemas corporativos. Geramos dados a todo o momento. Notícias, livros, e-mails, cadastros, logs, redes sociais, trânsito, dados meteorológicos, enfim, tudo o que a sociedade faz gera algum tipo de informação.

Em função dessa grande quantidade de dados e do pouco tempo disponível para fazer nossas tarefas, nasceram os sistemas de busca. A informação deve ser fácil de encontrar para ser útil. É nesse ponto que os buscadores tornam o trabalho mais produtivo, recuperando as informações necessárias para que as pessoas produzam mais. É uma situação que se aplica ao seu dia a dia, seja em uma empresa de tecnologia, escritório de advocacia, na universidade ou na hora de escolher o destino de uma viagem.

Os profissionais de programação podem imaginar um comando SQL como o *SELECT * FROM TABELA* quando falamos de sistemas de busca, mas esse recurso não é suficiente hoje em dia, quando o usuário está acostumado com buscadores no padrão Google.

Para um usuário comum, o entendimento é diferente. O cliente entra no site e procura por *pen drive*, *pendrive*, *flash drive* ou *memory stick*. Em todos esses casos, ele está procurando o mesmo produto. Igualmente pode acontecer com os nomes das pessoas. No Brasil, Francisco é chamado de Chico e Marco pode ser chamado de Marcos ou Marcus. Aposentadoria e pensão são termos equivalentes. Supermercado, hipermercado ou mercado também são palavras afins.

Some-se a isso a grande quantidade de dados digitais disponíveis, o chamado *Big Data*, e temos um cenário desafiador. Entregar as informações solicitadas pelos usuários em um curto espaço de

tempo (coisa de milissegundos), consultando gigantescas bases de dados da ordem de gigabytes ou terabytes. Algumas empresas já falam em petabytes.

Neste livro, vamos falar sobre sistemas de busca e as tecnologias usadas para resolver esse problema. Temos os termos *motor de busca*, *sistema de busca* e *buscador* como sinônimos, ou ainda *sistema de recuperação da informação* (SRI). Uma boa definição para o nosso trabalho de busca está no livro *Introduction to Information Retrieval* (Cambridge University Press, 2008):

> Recuperação da Informação (RI) é a busca por material (geralmente documentos) de natureza não estruturada (geralmente texto) que satisfaz uma necessidade informacional e está contido em grandes coleções (geralmente armazenado em computadores).

Características dos dados

Entenda que a linguagem humana é um tema complexo. O seu processamento com o uso de computadores, o que chamamos de *Processamento de Linguagem Natural* (PLN), é também difícil e os buscadores da internet tentam resolver esse problema. Acontece que a busca em texto tem que considerar muitas questões, como fonética, preposições, acentos, sinônimos, sugestões, apelidos, abreviaturas e figuras de linguagem.

Fica claro que o português (ou qualquer outro idioma) é uma língua complexa e é comum que o usuário escreva uma ou outra palavra incorretamente. Existem vocábulos difíceis como: assessoria, exceção, piscina, essencial, excesso, excelente, estender e extensão. São verdadeiras arapucas da linguagem e qualquer um mais apressado pode cair nela e escrever uma letra errada.

1.1 Linguagem natural

Até aqui falamos de conteúdo textual unicamente. Mas a comunicação vai um pouco além. A comunicação envolve uma mensagem que é enviada do emissor ao receptor através de um meio. Pode ser através da fala, de textos escritos em papel e, certamente, meios eletrônicos, sendo que neste último temos ferramentas e formas diferentes de nos comunicar.

Arquivos binários como um PDF, Word, Excel, mensagens instantâneas por SMS/WhatsApp, e-mails, posts de um blog, notícias de um portal, conversa no Skype... Cada tipo de texto tem características e vocabulário próprio. Quando escrevemos um memorando e um post de blog usamos técnicas e formatos diferentes.

Em todos os casos essa comunicação se dá através da *linguagem natural*. É como nós humanos nos expressamos: através de uma linguagem. As línguas são formadas de sintaxe e gramática e mesmo com essas otimizações tentativas de padronização, ainda comportam ambiguidades, o que contribui para o ruído na comunicação. É o que acontece quando você quer passar uma informação e a outra pessoa entende algo totalmente diferente.

Agora vamos contrastar com os computadores. Ao contrário do que parece, eles não são tão inteligentes assim. Um computador entende e processa muito bem linguagens formais, como equações matemáticas ou SQL. Aí é que mora o problema. Há uma grande distância entre linguagem natural e linguagem formal. O ser humano não entende muito bem linguagem formal e o computador não entende muito bem linguagem natural. A maioria da população (ainda) não consegue se comunicar em XML e o computador não sabe (ainda) escrever um livro.

É exatamente onde o *Processamento de Linguagem Natural* entra em campo. Esse ramo da computação tenta diminuir a distância entre homem e máquina. Não é uma tarefa simples. Por isso os

buscadores têm papel cada vez mais importante no nosso cotidiano. Um motor de busca tenta converter a linguagem natural de um ser humano em critérios objetivos que o computador entende. O usuário normalmente vai pesquisar *qual o melhor time de futebol?* e o motor de busca deve analisar estatísticas, notícias e outros dados para tentar encontrar a resposta adequada.

1.2 Sistemas de busca

Escrever sistemas de busca não é uma tarefa simples. Ele possui duas motivações: a flexibilidade e a velocidade da consulta. Primeiro vamos falar da flexibilidade, com a qual o usuário pode recuperar informações através de critérios complexos ou em parâmetros incompletos. Um critério complexo é quando você faz uma pergunta no Google ou Bing. *Vai chover hoje?*, *Qual a melhor operadora de celular?* e *Onde fica o Caribe?* são exemplos de perguntas não triviais que o buscador precisa trabalhar muito para responder.

Os buscadores são tão eficientes que o autocompletar já tenta fazer isso enquanto você digita. São esses os parâmetros incompletos da pesquisa. Você digita uma parte do critério de busca e o buscador tenta adivinhar o resto.

A segunda motivação é a velocidade, sendo que o resultado deve ser mostrado em milésimos de segundo. A velocidade é tão importante que até hoje o Google mostra o tempo que gastou para realizar suas buscas. Pode conferir, fica logo no começo da página de retorno.

Google, Bing, Yahoo!, Baidu, AOL (sim, ele ainda existe) e Ask são buscadores de uso geral. Não são os únicos, mas respondem pela quase totalidade das buscas na internet. Neste tipo de buscador, a

flexibilidade e velocidade são vitais e definiram os atores principais do mercado, Google e Bing. Em termos de precisão e tecnologia, eles são bastante eficientes, mas o domínio do Google é evidente.

Quando usar

Toda aplicação precisa de um mecanismo de busca. Em geral, é uma consulta SQL em um sistema gerenciador de banco de dados relacional (SGBDR). A proposta deste livro é mostrar alternativas a essa abordagem tradicional para construção de soluções de busca flexíveis e interessantes, com as funcionalidades que o usuário da internet está acostumado. A ideia é adicionar recursos para que o cliente consiga encontrar a informação que precisa de forma rápida.

O Apache Lucene, que é o tema central do nosso livro, é uma ótima opção para aplicações que geram muito texto, onde não podemos prever o tamanho ou o conteúdo da informação. Nesse caso, um banco relacional não tem mecanismos eficientes para recuperar as informações com velocidade e com a mesma flexibilidade que a linguagem de consulta do Lucene permite.

Você precisa de um motor de busca quando suas buscas são baseadas em texto e precisam suportar essas características:

1. Otimizado para busca textual de alta performance.
2. Esquema de dados flexível.
3. Suporte a ordenação de documentos.
4. Suporte para a escala da internet: otimizado para leituras.
5. Orientado a documentos.
6. Geolocalização.

Considere o Lucene também se a análise do texto é relevante e traz algum ganho. Nessa direção temos o *text mining*, que é uma área da inteligência artificial (IA) e, como tudo referente a IA, é muito

interessante. Não apenas pelo desafio técnico, mas também porque o resultado agrada ao usuário.

Para esses outros cenários onde as buscas são mais complexas que um simples SQL é que usamos ferramentas como o Apache Lucene.

1.3 Apache Lucene

O Apache Lucene é uma biblioteca de recuperação da informação para construção de sistemas de busca. Ele pode ser facilmente integrado a aplicações existentes e dá ao usuário o poder e comodidade de encontrar a informação desejada. Ao mesmo tempo, o programador tem uma API fácil de usar e já otimizada.

Com Lucene não é necessário criar índices nas tabelas, otimizar consultas gerenciar tamanho de *table space* ou se preocupar com a modelagem dos dados. Essas são tarefas de manutenção de um banco de dados relacional e não têm relação com mecanismos de busca como o Lucene.

O Lucene encontra rapidamente documentos e registros com base em critérios de pesquisa flexíveis. Esses documentos podem ser arquivos binários como Word/Excel/PDF, registros em um banco de dados ou sites da internet/intranet da empresa. Não faz diferença. A resposta é sempre muito rápida mesmo quando temos grandes volumes de dados. Você pode encontrar documentos em milésimos de segundo dentro de conjuntos de dados com milhões arquivos e com dezenas de gigabytes de tamanho. Esse é o trabalho do Lucene.

A forma como fazemos buscas sempre foi limitada pelos recursos computacionais disponíveis. E temos cada vez mais recursos. Agora é o momento de converter todo esse poder de processamento em ferramentas com mais possibilidades de atender às demandas dos

usuários, até mesmo demandas que os usuários sequer sabem que existem. Com Lucene, podemos ter um sistema de busca como o Google dentro da nossa aplicação.

São muitas novas possibilidades e oportunidades, como criar sistemas de busca para o usuário interno da empresa, porque o usuário de um sistema de busca não é somente o cliente externo. Podemos criar um módulo de busca para os administradores e o pessoal do suporte. Quantas vezes você já tentou encontrar um registro específico no meio de milhões de linhas de log? Uma ideia seria criar uma aplicação que encontre as páginas mais acessadas, fornecendo uma visão estratégica do comportamento dos sistemas, servindo também para auditoria.

Outras características interessantes são a ordenação e o *faceting*. Podemos definir critérios de relevância e mudar a ordem em que aparecem os registros, trazendo os itens mais importantes no começo do resultado.

Faceting é o agrupamento do resultado da busca em categorias baseadas nos termos indexados. Isso ajuda o usuário a explorar o resultado da busca de acordo com as categorias de seu interesse. Considere uma loja de livros. As categorias podem ser preço, autor e data de publicação. O cliente pode navegar apenas nos itens que interessam. A seguir estão listadas algumas situações onde o Lucene pode ajudar:

- Quando o usuário precisa realizar consultas complexas em grande volume de texto. Neste caso, o comando `like` do SGBD não é suficiente.
- Quando há uma grande quantidade de acessos simultâneos. Mesmo com otimizações, consultas em SGBDR consomem muitos recursos computacionais.
- Quando seu sistema precisar de um módulo de consulta pela internet. Neste caso, não temos como controlar a quantidade de acessos nem a quantidade de registros retornados.

- Quando as consultas no SGBDR não têm um tempo de resposta aceitável.

Nestes casos, o Lucene é uma boa opção porque podemos criar ferramentas de busca eficientes com baixo custo de hardware. Esse custo é baixo, claro, quando comparado às soluções de alta performance dos grandes fabricantes. Para atender a milhões de acessos com alta disponibilidade não é difícil ter de adquirir soluções que custam alguns milhões de reais.

Principais características do Lucene

Inicialmente escrito em Java, o Lucene foi portado para diversas linguagens, os chamados *ports*. Assim, o Lucene está disponível em linguagens como C, C++, Objective C e .NET. As características descritas no livro se referem apenas à versão em Java. Dessa forma, as características do Lucene são:

- Grande velocidade de busca: mesmo que você tenha um conjunto de documentos com milhões de páginas, uma busca não leva mais que milésimos de segundo;
- Escalabilidade: a base de dados pode aumentar para milhões ou bilhões de documentos e a performance será garantida;
- Facilidade de integração com sistemas já existentes: dispõe de uma API simples, com poucas classes e métodos que podem ser utilizados em qualquer sistema;
- Linguagem de consulta flexível: a linguagem de consulta do Lucene foi desenvolvida para atender às demandas mais complexas dos usuários;
- Possui diversos projetos relacionados para processamento de informação, como o Hadoop, Mahout e OpenNLP.

1.4 Muito mais do que apenas buscas

Sistemas de busca não são mais desenvolvidos apenas para busca. Os usuários já estão acostumados com a correção ortográfica, a busca por similaridade e até o que conhecemos como "você quis dizer". Quando você digita o nome de uma pessoa no Google, ele não mostra apenas o resultado da busca, mas também a entrada da Wikipedia, imagens, notícias e pessoas similares. É o que chamamos de *named entity recognition* (NER). O buscador reconhece no conteúdo de um texto entidades como pessoas ou lugares e mostra as informações que têm alguma relação com eles.

Se você usa o Google Maps durante uma viagem, ele sugere os restaurantes mais próximos, bem como as opções de lazer. Quando você vai para casa no fim de um dia de trabalho, o Google avisa que o tráfego está lento. Tudo isso é feito através da análise dos dados registrados durante anos, que é usado para conhecer seu perfil, onde você mora e o que gosta de comer. No meu caso, que tenho dois filhos, ele também sugere atividades com crianças.

Dessa forma, quando fazemos uma busca no Google, na verdade o que acontece é muito mais do que apenas a procura dos itens relevantes para aquela necessidade imediata de informação. Este é o estágio atual dos sistemas de busca.

Considere outra situação, como os dados gerados pelo *call center* da sua empresa. Cada atendimento gera um documento que contém a interação com o usuário. E cada operador classifica o incidente de acordo com uma lista limitada de opções, como dúvida, reclamação, pendências financeiras etc. Esse tipo de informação é interessante em um sistema de *business intelligence*, para extração de dados estatísticos. Dessa forma você pode descobrir qual a quantidade de reclamações de um produto. É uma análise apenas quantitativa de ocorrências.

Com o Lucene, claro, você pode achar facilmente qualquer registro, porque esse é um dos objetivos de um sistema de busca. Mas podemos ir um pouco além. Um motor de busca pode agrupar conteúdo semelhante. E o que isso significa? Para um gerente seria

interessante, por exemplo, saber quais reclamações são similares, com base no relato do usuário. Essa informação pode levar a empresa a ser proativa e corrigir o problema antes que ele gere mais prejuízo. Da mesma forma, pode-se agrupar elogios sobre um novo serviço e descobrir se o usuário está satisfeito com a empresa.

Se o cliente é uma loja de roupas online, você poderia criar uma aplicação para analisar as opiniões dos clientes que são publicadas nas redes sociais ou no próprio site. Dessa forma, você poderia saber se a coleção foi bem recebida pelos compradores.

Um portal de notícias poderia usar o histórico do usuário para mostrar as notícias em que ele tem mais interesse. Eu tenho interesse por assuntos ligados a tecnologia e ciência e você pode gostar de esportes. Para mim, conteúdo ligado a esporte é menos relevante que conteúdo ligado a ciência. A disposição das notícias deve seguir essas preferências para manter a audiência do portal. E esse processamento deve ser dinâmico. Imagine o caso de uma pessoa que acaba de casar. O sistema deve ser atualizado para identificar essa mudança na vida do usuário e entregar o conteúdo adequado.

Recentemente vimos o aparecimento e popularização dos sites de viagens. É improvável que você viaje hoje sem consultar esse tipo de site. São aplicações que fazem uso intensivo da geolocalização e os sistemas de busca incorporaram essa funcionalidade. Se estamos em outra cidade, a localização do estabelecimento deve ser levada em consideração. Caso queira comida italiana em São Paulo, o ideal é que sejam mostradas as opções levando em consideração a distância em relação à sua posição.

Encontrabilidade e os sistemas de busca

A informação deve ser encontrável, não basta estar disponível. E a aplicação deve garantir que será fácil para o usuário operar os buscadores. Até mesmo uma criança tem que conseguir encontrar o que procura e as ferramentas devem se desenvolver cada vez mais

para entender o que o usuário precisa e interpretar a busca, encontrando a resposta correta para cada demanda.

Os buscadores fazem parte do cotidiano da sociedade. Crianças e adultos, técnicos ou não. Usar o Google e o Bing é uma ação natural no trabalho, na escola, para fazer compras, para escolher uma televisão e para encontrar um emprego.

Sistemas de busca não são apenas para texto. Atualmente, a busca por imagens, áudio e vídeo é tão ou mais importante. O YouTube é um dos buscadores mais utilizados na internet e está transformando a forma como consumimos informação. Em um passado não muito distante perguntávamos para nossa mãe sobre uma receita de bolo. Hoje você pode encontrar canais de culinária com diversas alternativas e todos tipos de pratos que se pode imaginar.

Tipos de usuário

Há dois tipos de usuário em um sistema de busca. Um sabe exatamente o que procura e há um segundo que não sabe. A maioria dos usuários é do segundo tipo, mesmo que seja um sistema de busca especializado.

O primeiro grupo é daquelas pessoas que usam os recursos avançados dos sistemas de busca e sabem quais os sites que contêm a informação. É quando você digita várias palavras-chaves com operadores lógicos (AND, OR, NOT) no buscador, refinando a pesquisa. São geralmente técnicos ou especialistas na área. E são a minoria.

O segundo tipo de usuários é aquele que não tem ideia de como achar a informação. Ele digita geralmente um termo no buscador, como "qual a melhor televisão", e espera encontrar dicas dos melhores aparelhos e as lojas que vendem com o melhor preço. Nesses casos o buscador tem que analisar os sinônimos e a grafia e tentar entender o que o usuário quer de verdade. É para esse grupo que construímos as ferramentas mais interessantes de busca.

1.5 O que um motor de busca não faz

É importante saber o que um sistema de busca faz, assim como é importante saber o que ele não faz. Ou o que ele não faz muito bem. Sabendo das limitações podemos focar nos pontos fortes das ferramentas.

O Lucene é útil em situações em que a operação dominante é a consulta. Tenha sempre isso em mente. E ele foi projetado para retornar um conjunto limitado de registros, geralmente 100 itens, que é mais do que suficiente na maioria das situações. Para retornar quantidades maiores, você pode usar o recurso de paginação do capítulo *7. Integração com sistemas corporativos*.

Para situações em que a principal operação é a inclusão ou alteração de registros, o Lucene pode não ser a melhor escolha porque não tem uma boa performance para gravação. A cada versão, a velocidade de gravação é otimizada, mas ainda assim não é comparável com um banco NoSQL (Not Only SQL). Em todos os casos, estamos falando de operações em lote, com milhões de itens sendo inseridos ou consultados.

Não significa que o Lucene deve ser evitado em sistemas com grande volume de gravação. O Lucene pode inclusive funcionar como um banco de dados não relacional.

1.6 Ecossistema do Lucene

As atualizações no mundo *open source* são rápidas e constantes. Comecei a escrever esse livro com o Lucene 4 e agora está na versão 5. A evolução não para e a lista de mudanças é extensa.

A melhor forma de acompanhar as novidades no Lucene é através das listas de discussão da Apache. Há dezenas delas, divididas

entre grupos de usuários e de desenvolvedores. A lista de discussão de todos os projetos é http://mail-archives.apache.org/mod_mbox/ e há uma seção específica para o Lucene.

No grupo de usuários podemos fazer perguntas sobre como usar o Lucene. No geral, são perguntas específicas e com um nível mais avançado, mas também encontramos questões básicas, de quem está começando com a ferramenta. Está disponível em http://mail-archives.apache.org/mod_mbox/lucene-solr-user/.

O grupo de desenvolvedores trata sobre a implementação de novas funcionalidades ou correção de *bugs*. A lista está dividida por mês e ano. Está disponível em http://mail-archives.apache.org/mod_mbox/lucene-dev/.

O site do *Jira* da Apache está disponível em https://issues.apache.org/jira/ e nele você acompanha as evoluções e correções do projeto Lucene. Também é possível relatar erros, propor melhorias e escrever o código das correções, que será avaliado por uma equipe antes de ser incorporado ao projeto.

1.7 Aplicações de exemplo e código-fonte

O código-fonte usado nos exemplos do livro utiliza as melhores práticas de programação sempre que possível. Em alguns casos optei por escrever código mais objetivo, ignorando conceitos de Orientação a Objetos ou normalização de bancos de dados. Isso é necessário porque não estamos tratando de dados relacionais.

Para ilustrar os exemplos foram criados alguns projetos no GitHub. O objetivo é incluir a maior quantidade possível de ideias, para as mais diversas situações. O código-fonte está disponível nos seguintes repositórios:

- https://github.com/masreis/exemplos-livro-lucene/: código-fonte dos exemplos discutidos no livro. Mostra uma visão teórica dos conceitos e sua aplicação no código;
- https://github.com/masreis/e-commerce/: aplicação web de e-commerce que usa o Lucene para ilustrar os exemplos do livro;
- https://github.com/masreis/e-commerce-hibernate-search/: aplicação web de e-commerce o Hibernate Search para ilustrar os exemplos do livro.

O código-fonte deste livro foi utilizado em ambientes de homologação e produção, além de ter sido testado em sistemas operacionais diferentes (Linux, Mac e Windows). Entretanto, sabemos que erros acontecem e até fazem parte do processo. Se você encontrar um erro, tanto no texto como no código, seria ótimo se você pudesse nos informar. Os dados de contato estão disponíveis em http://marcoreis.net/, ou diretamente em ma@marcoreis.net.

Capítulo 2
Conceitos de recuperação da informação

Este é um livro técnico, prático e objetivo, com código-fonte e exemplos direcionados para solução de problemas reais. Entretanto, por se tratar de um tema muito específico, precisamos explicar seus conceitos. É exatamente para isso que temos este capítulo, onde vamos definir os termos mais comuns que serão usados no decorrer do nosso trabalho.

Os conceitos e ideias mencionados são objeto de várias pesquisas e podem ser aplicados também nas empresas. Temos ideias que são simples, como a busca em texto livre, e outras mais complicadas, como o TF-IDF. Não se preocupe, veremos tudo isso mais adiante. O Lucene implementa esses conceitos através da sua API.

Existem muitos módulos, para funções mais complexas do que apenas busca. Lembre-se de que os sistemas de busca não são mais apenas para busca. A cada nova versão do Lucene são adicionados novos módulos, incluindo questões como linguística, novos algoritmos de busca, classificação, faceting etc.

Hoje, o Lucene é composto das bibliotecas que estão listadas logo a seguir, mas nada impede que sejam adicionadas novas funcionalidades em versões futuras. As principais classes do Lucene, as que formam o seu núcleo, estão nos pacotes *core*, *analyzers-common* e *queryparser*. Os demais pacotes são responsáveis por funções de suporte e refinamentos não diretamente ligadas diretamente à busca. São funcionalidades como análise fonética, análise linguística (Apache UIMA) e classificação de documentos. Foram omitidas da lista as bibliotecas para línguas como polonês e japonês porque não são muito úteis para a maioria das aplicações.

- *core:* classes de base do Lucene;
- *analyzers-common:* analisadores para diferentes linguagens, incluindo português;
- *analyzers-icu:* para integração com ICU (International Components for Unicode);
- *analyzers-phonetic:* analisadores para fonética;
- *analyzers-uima:* integração com Apache UIMA;
- *benchmark:* sistema para *benchmarking* do Lucene;
- *classification:* usa documentos já indexados para classificar novos itens.
- *codecs:* codecs e formatos;
- *demo:* código com exemplos de aplicações;
- *expressions:* processamento dinâmico de valores baseados em gramática;
- *facet:* indexação e busca facetada;
- *highlighter:* utilitários para fazer *highlight* do resultado de buscas;
- *memory:* permite indexação em memória, em substituição ao anterior recurso de `RAMDirectory`;
- *misc:* utilitários para indexação;
- *queries:* alguns filtros e consultas, como a `MoreLikeThisQuery`, uma consulta que retorna itens similares;
- *queryparser:* analisadores de consulta;
- *replicator:* utilitários para replicação;
- *sandbox:* contribuições de terceiros e novas ideias;
- *spatial/spatial3d:* busca geoespacial;
- *suggest:* sistema de sugestão e correção ortográfica;
- *test-framework:* para testar aplicações Lucene.

Realmente são muitas funcionalidades e para ver o que cada uma faz nós usaremos os projetos de referência descritos no capítulo *1. Introdução*, principalmente o projeto *e-commerce*, que é a aplicação dessas funcionalidades em um cenário real.

2.1 Recuperação da Informação

Recuperação da Informação (RI) diz respeito ao ato de selecionar itens relevantes dentro de uma coleção de documentos. Essa relevância considera a necessidade de informação do usuário, representada através de uma consulta. É natural que neste primeiro momento alguns termos não estejam claros. Algumas expressões são mais conhecidas pelos profissionais da busca, como os especialistas em biblioteconomia. Outras definições são técnicas, mas ainda assim pouco comuns para a formação tradicional na Ciência da Computação.

Sua importância vem aumentando, principalmente porque é mais prático para o usuário usar um sistema de recuperação da informação para acessar dados como e-mail ou histórico de vendas. Até pouco tempo atrás você precisaria anotar o número do pedido da compra para ser atendido pela loja. Aliás, algumas (poucas) ainda exigem isso, mas é certamente menos comum. Hoje, você só precisa informar o nome ou CPF. Com base nisso é possível recuperar todo o seu histórico e analisar o status do seu pedido. Muito mais fácil, não?

Para entender os conceitos associados à RI (Recuperação da Informação) e ao Lucene, vamos dar uma olhada em vários termos usados nesta área. Os principais são índice, documento, campo e termo. Depois disso veremos outros conceitos mais avançados, usados na construção de sistemas de busca mais complexos.

2.2 Índice

A base do Lucene é o índice. Para a área da Recuperação da Informação, índice é uma estrutura que serve como guia para encontrar itens específicos em uma coleção de documentos. Depois de criado, podemos realizar consultas no índice para recuperar os

documentos que o usuário precisa. O índice pode funcionar como uma biblioteca. O usuário (a aplicação) acessa a biblioteca (o índice) e solicita ao bibliotecário (a API do Lucene) um livro (ou documento). O bibliotecário conhece bem a biblioteca e pode encontrar rapidamente o que o usuário precisa, mesmo quando existirem muitos documentos.

Para o Lucene é um pouco diferente. O índice é um diretório onde são guardados os dados binários sobre os documentos, como os campos (metadados), ponteiros, dicionários e exclusões. Esses tipos de arquivos e suas configurações serão vistas no capítulo *5. Principais classes*.

Por ora, o que precisamos entender é que o índice do Lucene é o diretório que armazenará os documentos indexados. Os documentos serão vistos na próxima seção.

O que deve ser indexado

Devemos criar o índice de acordo com o tipo de busca que o usuário vai realizar, tendo em mente que este índice deve conter toda informação necessária para montar o resultado adequado. Assim, devemos indexar todos os dados que podem ser consultados.

Por exemplo, vamos criar um motor de busca para indexar os dados de vendas de um sistema de e-commerce. Quando o usuário fizer uma consulta, o índice deve conter todos os dados necessários para montar a tela com o resultado da busca, não apenas os códigos ou chaves primárias das tabelas. A consulta ao Lucene deve retornar todos os dados referentes à venda. Então, o índice deve conter os dados completos do cliente (CPF, nome e endereço), os dados do produto (código, nome, categoria e preço) e os dados da venda (valor pago, data do envio e da entrega).

Veja que o cliente pode pesquisar por qualquer um desses campos e essas informações podem ser mostradas no resultado da busca. E se o usuário quiser pesquisar pela forma de pagamento? Como no

caso de um gerente que precisa saber a faixa de vendas que são feitas pelo cartão de crédito. Note que a forma de pagamento não foi indexada, assim, não podemos pesquisar por esse critério. Neste caso, o índice deve ser recriado, acrescentando essa nova informação. Mais detalhes serão vistos no capítulo *3. Indexação e busca*.

Com essa estrutura, não será necessário realizar consultas extras no banco de dados ou outro mecanismo de armazenamento. Ora, se a ideia aqui é velocidade de resposta (baixa latência), misturar duas tecnologias (Lucene e SGBDR) não será muito performático. Mesmo com toda a velocidade do Lucene, se você misturar as duas tecnologias, vai ter de esperar pelo retorno do banco de dados, que tende a ser mais lento.

Agora vamos considerar uma empresa com vários sistemas, como o sistema de RH, financeiro, fiscal, contabilidade e *workflow*. O mais interessante seria ter um índice para cada sistema, cada um com seu diretório separado.

Aqui vale uma dica. Quando o usuário fizer uma consulta no motor de busca, este deve retornar todas as informações necessárias para entregar o resultado completo e, se possível, formatado. Isso significa que o motor de busca não deveria realizar consultas no banco de dados para buscar outras informações. Tudo deve estar armazenado no índice e todos os campos necessários para montar o resultado para o usuário devem estar indexados.

Por exemplo: não guarde no índice apenas o CPF da pessoa ou o código do produto para depois consultar o nome ou a descrição no banco de dados. Pelo contrário, guarde todos os campos necessários para montar o resultado da consulta de volta para o usuário. Armazene no índice o CPF e o nome completo, bem como o código e a descrição e preço do produto. Guarde todos os dados que poderão ser usados futuramente na pesquisa. O índice não deve ter a preocupação da normalização dos dados, assim como no modelo relacional de um banco de dados tradicional.

Vamos para os exemplos. Em uma loja online, você poderia ter mais de um índice para demandas diferentes. Veja:

- *Índice para produtos*: este é um ponto crítico. Mesmo que a quantidade de produtos não seja grande, precisamos do Lucene para encontrar a maior quantidade possível de itens para a consulta do usuário. Se ele digitar *pen drive*, *pendrive*, *pendriver*, *flash drive*, ou qualquer coisa nesse sentido, temos de conseguir mostrar a melhor resposta. Se o usuário não encontrar o que precisa, ele vai para outro site. Se for um portal de notícias com um sistema de buscas ineficiente, ele perde acessos.
- *Índice para histórico de vendas*: o histórico de vendas em um site de e-commerce costuma ser grande. Um único produto pode gerar milhares de pedidos. O usuário do sistema de buscas, que pode ser o cliente externo ou o gerente da loja virtual, deve ser capaz de encontrar esse pedido de várias formas, não apenas pelo código de venda. Aliás, essa é uma das maiores queixas que ouço dos clientes: não conseguir encontrar os itens através de qualquer campo que ele conheça. Isso é facilmente resolvido com Lucene. Os campos que você guardar no Lucene são pesquisáveis e o usuário poderá consultar por qualquer um deles.
- *Índice para log do servidor web*: o log do servidor web guarda o histórico de navegação do usuário no site, uma informação valiosa para avaliar quais as páginas mais acessadas ou quais os produtos mais pesquisados. Funciona como um termômetro das ações dos usuários no site. Com o Lucene você pode descobrir o que o usuário está fazendo para que o administrador do site tenha faça alguma ação, como promover áreas pouco acessadas. O Lucene torna possível inclusive fazer auditoria no sistema, uma atividade importante e que normalmente exige um tempo grande para ser realizada. Com o Lucene você encontra facilmente o histórico de acesso e de ações de um usuário no sistema.

2.3 Documento

Documento é a menor unidade de informação para um sistema de busca e é a principal entidade usada no resultado de uma consulta do usuário. Um índice é composto de vários documentos que, depois de indexados, estão disponíveis para as consultas dos usuários.

Exemplos de documentos são páginas web, registros de um banco de dados, catálogos de produtos, logs de servidores, histórico de buscas, histórico de vendas, enfim, qualquer informação que precisa ser recuperada futuramente.

No nosso sistema de e-commerce teremos vários documentos. Vamos começar com o índice de produtos. Cada produto será um documento. A *TV LCD LG modelo ABC-2016* será um documento. O tênis *Nike Air DEF-2017* será outro documento dentro do índice de produto. Depois vamos para as vendas e teremos um índice separado com vários documentos, cada um representando uma venda.

Um sistema de busca funciona seguindo essa sequência: o usuário digita um termo que pretende pesquisar na interface de busca do site de e-commerce, algo como "celular android", e o buscador deve retornar uma lista com os documentos mais relevantes com os termos "celular" e "android".

Tomando como base o mundo digital, um documento pode conter o conteúdo de um PDF ou de um arquivo do Word (DOC, DOCX, ODT), os metadados de uma imagem ou vídeo, o log de uma aplicação e também o conteúdo de uma tabela de banco de dados.

Para ilustrar o que seria um documento na prática, considere um sistema de e-commerce, como a que temos de referência no livro. Uma entidade que existe em qualquer sistema de compras, seja ele virtual ou loja física, seria o *produto*. É unicamente para isso que existe uma loja: vender produtos. Então, quando o usuário acessa o

site, ele provavelmente vai procurar por produtos. O sistema de e-commerce deve ter, então, um índice para armazenar os documentos associados aos *produtos*. Cada produto é guardado como um documento e cada documento guarda informações de apenas um produto. Teremos então um índice com vários documentos para representar esses produtos.

O Lucene não tem nenhuma restrição ao que colocamos no índice. Uma boa ideia é que cada índice contenha apenas um tipo de informação, como sugerido anteriormente neste capítulo. Os produtos ficam em um índice, o histórico de vendas, em outro índice, e o índice do log do servidor web deve também estar no seu próprio índice. E cada índice tem o seu diretório, só para lembrar.

Coleção de documentos

Coleção de documentos é o agrupamento de todos os documentos que você pretende indexar. É um conjunto de informações similares e úteis para nossa aplicação de e-commerce, como a coleção de documentos dos produtos e a coleção de documentos do histórico de vendas.

Assim como o índice, cada coleção de documentos deve conter dados sobre um e apenas um tipo de informação, como informações sobre os funcionários da empresa, dados sobre pacientes e dados sobre clientes, quando aplicado a outros sistemas.

Na verdade o Lucene não impede que você misture coleções de documentos. É bem possível fazer assim. Mas certamente não seria uma boa ideia misturar em uma única coleção de documentos tipos de informações diferentes. Então, uma coleção de documentos deve se referir a apenas um assunto por uma questão de organização dos seus dados.

2.4 Campo

Cada documento é formado por uma coleção de campos (*fields*). O campo tem nome, tipo de dado e valor, que pode ser alfanumérico ou numérico. Ele representa uma área para indexação (a informação armazenada no campo) e busca (através do nome do campo).

No Lucene o índice não tem restrição de campos. É permitido ter em um mesmo índice documentos com campos diferentes. Em um índice você pode ter documentos com os campos nome, descrição, fabricante e preço para representar um produto. Neste mesmo índice, quer dizer, gravado neste diretório, é possível ter outros documentos com os campos `dataDaVenda`, `valorTotal`, `nomeCliente` e `endereçoDeEntrega`, representado uma venda. Na hora de indexar, o Lucene não encontra nenhum problema. Na busca também não há problemas. Quando você pesquisar pelo campo `dataDaVenda` igual a `2016-01-01`, o Lucene retornará apenas documentos que tenham esse campo com o valor determinado. Sem problema, você recuperou as vendas daquele dia. Se por outro lado você pesquisar pelo campo `fabricante` igual a `Samsung`, teremos uma lista de produtos produzidos por essa empresa.

Pessoalmente, acho essa liberdade um problema para a manutenção do sistema. A partir do momento em que indexamos documentos muito diferentes, isso pode gerar confusão entre os programadores, além de que é provável que você precisará implementar mecanismos de controle (como *flags*) para que os documentos não se misturem.

Vamos considerar que temos um índice para os *produtos* disponíveis no nosso site de e-commerce. De quais campos precisamos?

Ora, precisamos de todos os campos que serão usados para buscar um produto ou para montar o resultado da busca para o usuário. Se tem alguma dúvida, pense assim: você vai precisar consultar produtos por um campo ou mostrar esse mesmo campo em uma

tela de resultado de consulta? Se a resposta é "sim", então esse campo deve estar no documento *produto*. Se a resposta é "não", provavelmente este campo não precisa existir.

Um documento completo com todas as informações do produto é mais útil para a busca e para mostrar o resultado ao usuário. Essas são as medidas que devem ser consideradas quando definimos a sua estrutura. Essas técnicas serão vistas com mais detalhes no decorrer dos capítulos, sempre adicionando novas dicas.

A seguir temos uma lista com os campos para um documento que armazena os produtos do nosso projeto de e-commerce.

- `ID` : identificador (obrigatório) do produto.
- `Nome` : busca mais direta, onde o usuário sabe o nome do produto.
- `Fabricante` : nome do fabricante. Imagine que há casos em que o usuário conhece apenas o fabricante de um produto.
- `Especificacao` : o usuário quer encontrar produtos que contenham características específicas, como HDMI.
- `Preco` : para filtrar os produtos por faixa de valor. Eu gostaria de comprar uma televisão que tenha HDMI e custe entre $1.000,00 e $1.200,00.
- `IDCategoria` : identificador da categoria.
- `Categoria` : um produto pode estar em várias categorias. Um tênis pode pertencer ao departamento de corrida, escalada, futebol, skate e musculação. Outra situação interessante é quando o usuário quer conhecer mais sobre uma categoria como *adega*, onde estariam produtos como vinhos, abridores, tampas e taças.
- `DataAtualizacao` : data em que o registro foi atualizado. Esse campo é usado para controlar a versão do documento.
- `DataIndexacao` : data em que o registro foi indexado. Usado para manter o índice atualizado.
- `ConteudoCompleto` : podemos criar um campo que contém o conteúdo textual de todos os outros campos, o que seria equivalente a uma busca livre.

> Todo documento deve ter um campo ID para identificá-lo unicamente e não é necessário que seja um valor numérico.

Como visto na lista de campos, devemos armazenar todos os dados referentes ao produto, incluindo os dados dos relacionamentos, como no caso da categoria. Veja que estamos incluindo o ID e a descrição da categoria, sem preocupação com a duplicação dos valores. Considerando que exista uma tabela ou um índice para categoria, esses dados estarão duplicados em todos os lugares que devem ser pesquisados. É possível fazer a junção de campos, como se fosse um *JOIN* do SQL, mas é um custo desnecessário, ao menos nesse nosso caso.

2.5 Termo

Termo representa uma palavra do texto, que é a unidade de busca do Lucene. É composto do nome de um campo e da palavra (ou palavras) que desejamos buscar. Um termo pode conter uma palavra-chave, uma data, um ID, e-mail etc. Quando o usuário quer consultar o índice, ele vai informar o termo que pretende encontrar, por exemplo, a busca pelo produto *pen drive* será executada com o termo `Nome:(pen drive)`.

Para uma busca mais precisa, pode-se combinar os termos utilizando operadores lógicos (AND, OR e NOT). Então, o exemplo anterior pode ser mais específico assim: `Nome:(pen drive)` *AND* `Fabricante:sandisk`. Agora o cliente procura o *pen drive* de um fabricante específico. É uma consulta mais restritiva. Veja que o termo de consulta pode conter uma única palavra ou pode conter várias palavras delimitadas por aspas duplas. Um exemplo seria o termo `Nome:"pen drive"`. É um único termo que contém duas palavras.

A consulta mais básica no nosso sistema de e-commerce é quando o cliente acessa a busca e digita o termo `nome:tv`. Significa que ele quer encontrar os produtos que tenham o nome *TV* (televisor). Mas pode ser mais que isso. Um termo pode ser `nome:(tv monitor)`. Agora, o usuário quer encontrar os documentos com nome *TV* ou *monitor*. Essa é a linguagem de consulta do Lucene, que será exaustivamente usada durante o livro.

2.6 Fases de processamento

Um sistema de busca tem duas fases bem definidas: a *indexação* e a *busca*. Para a busca funcionar, temos que indexar os documentos. Essa é a primeira fase. A indexação é a ação de construir o índice e tornar os documentos encontráveis.

Dada uma coleção de documentos, como a lista de produtos disponíveis no nosso site, vamos coletar cada um desses itens e indexá-los com o Lucene. A partir desse momento o usuário pode buscar por esses produtos. A origem dessa coleção de documentos não importa para o Lucene. Os produtos podem estar em um arquivo texto, em PDF, no banco de dados etc. O que importa mesmo é que você vai escrever uma aplicação que leia cada um desses documentos e escreva no índice. O histórico de vendas é um pouco diferente, porque ele simplesmente não existe até que seja feita uma venda. Após o usuário completar a venda é que o documento respectivo será escrito no índice. O nosso último índice será o de logs da aplicação. Esse é o maior, pois contém cada um dos cliques no site, incluindo a navegação do usuário com cada página que ele acessou.

Para o usuário final, a fase mais importante é certamente a busca, que encontra os documentos correspondentes e mostra o resultado para uma consulta. Na busca, o usuário vai fornecer os termos que serão consultados e o Lucene vai devolver uma lista de documentos

que contêm aqueles termos. O primeiro caso é quando o usuário é um cliente à procura de um produto, o segundo caso é um usuário à procura de uma venda realizada e a terceira opção será para auditoria, onde você poderá ver o que cada usuário fez no sistema.

Note que não estamos falando de processamento em tempo real. Os dados não estão disponíveis instantaneamente quando são cadastrados. A indexação dura alguns segundos, ou até minutos. Somente após a atualização no índice é que o documento estará disponível para a consulta do usuário. É o que chamamos de *Near Real Time* (NRT), ou sistema em tempo quase real. A operação é assíncrona e será concretizada um pouco depois de ser iniciada.

Na verdade, não são todos os sistemas onde o processamento é feito em tempo real. Veja o sistema bancário nacional. Quando você efetua uma transação bancária, ela é agendada para execução em uma fila com milhões de itens (outras transações). Na maioria dos casos é uma transação rápida, mas internamente ainda estamos falando de uma fila. É uma operação assíncrona, estratégia muito comum em integração de sistemas.

Indexação

Antes da busca do usuário, os servidores do Google estavam indexando a internet e os buscadores (Google, Bing, Ask etc.) estão a todo momento fazendo isso. E não apenas o conteúdo textual dos sites. Imagens, vídeos, notícias, blogs e fóruns, tudo é indexado pelos buscadores. Por isso a sua busca é tão rápida.

Quando você faz uma busca na internet com o Google, na verdade é feita uma busca nos servidores do próprio Google, que anteriormente indexaram toda a internet. Isso significa que o Google (ou outro buscador como o Bing) fez uma cópia da internet inteira, de todas as páginas disponíveis e as indexou em seus servidores. Sim, são muitos servidores para conseguir fazer essa tarefa, centenas de milhares de computadores espalhados no mundo. Para

concluir, a sua busca é feita nos servidores do Google (ou Bing) e não na internet, como poderíamos supor.

Com o Lucene usa-se o mesmo princípio. Inicialmente, precisamos indexar os documentos que estarão posteriormente disponíveis para busca. Dessa forma, considerando as definições de diversos autores, vamos definir que:

> A indexação é o processo de transformar documentos em objetos encontráveis por meio da busca.

A indexação compreende basicamente quatro tarefas:

1. Coletar os documentos que serão indexados.
2. Dividir o texto em palavras.
3. Realizar o pré-processamento linguístico das palavras.
4. Criação do índice, com os documentos e seus termos.

O primeiro passo é definir quais textos serão indexados. Pode ser um banco de dados, um site, um diretório de arquivos PDF etc. É possível indexar a maioria dos formatos digitais disponíveis. Essa será nossa coleção de documentos.

Na segunda tarefa do processo de indexação, as palavras do texto são separadas, para logo em seguida serem analisadas. Na terceira fase ocorre a análise linguística. É o que chamamos de pré-processamento, ou normalização. As letras são transformadas em minúsculas (`Casa` é convertida em `casa`), é definida a morfologia (substantivo, artigo, adjetivo, numeral, pronome ou verbo) e ocorre a remoção das palavras sem relevância e dos caracteres especiais. O Lucene tem um analisador que faz uma limpeza nas palavras antes de indexar. O analisador para o idioma português faz mais algumas tarefas, como retirar acentos, tratar sufixos, plural, gênero (feminino e masculino), advérbio, aumentativo, substantivos, verbos e vogais de acordo com a nossa gramática.

Por isso, a busca por TV (letras maiúsculas) é, na verdade, a busca pela palavra normalizada tv (minúsculas). O mesmo vale para micro-ondas e Blu-Ray: após a normalização padrão teremos os termos *micro ondas* e *blu ray*, ambos sem hífen e com letras minúsculas. Esses detalhes serão vistos no capítulo *3. Indexação e busca*.

Exemplo: considere que um sistema de buscas para dados médicos contém termos específicos para a área. Sua lista de sinônimos é diferente de um texto jornalístico. As palavras *ensaio*, *teste* e *estudo* são sinônimas, mas em um texto de tecnologia isso não é verdade. Existe uma família de medicamentos chamados de aminas secundárias, como a nortriptilina, desipramina e lofepramina. Além deste, há outra família de aminas, as terciárias, como a amitriptilina e dotiepina. Estes igualmente sinônimos. A frase *vitaminas A e E* diz respeito à *vitamina A* e *vitamina E*, contudo, o analisador padrão do Lucene não entende assim, o que diminui a precisão da busca, claro, neste caso específico. Um caso linguístico mais complicado é a *Doença de Alzheimer*, chamada simplesmente de *DA*. Veja que *DA* é, da mesma forma, uma *stop word*, segundo nossa gramática é a contração da preposição *de* e do artigo *a*. Como toda *stop word*, sua relevância é baixa. Isso significa que seu peso no cálculo de relevância é baixo.

Após a análise e tratamento dos termos, o texto é transformado em um documento do Lucene, um arquivo binário e proprietário que será gravado no diretório do índice. Para clarificar, documento não é um arquivo do Word ou PDF. Estamos usando a definição de documento para sistemas de recuperação da informação. Assim, neste livro, documento é qualquer conteúdo textual que precisa ser indexado.

Há mais alguns detalhes para melhorar a qualidade do sistema de busca, que estão disponíveis geralmente em produtos pagos. Por exemplo, a separação do texto através de unidades significativas e não apenas em palavras. É o caso de termos compostos, como *São Paulo*, ou *São José do Rio Preto*. Mesmo tendo mais de uma

palavra, representam apenas uma unidade significativa. Em sistemas mais complexos precisamos analisar esses casos.

Apenas após a indexação o índice estará construído e o usuário pode realizar buscas.

Busca

A funcionalidade mais importante para um buscador é, naturalmente, a busca. O usuário deve encontrar o que precisa com facilidade, velocidade e a maior precisão possível. E o resultado dessa busca deve trazer itens significativos (ou relevantes). Nesse aspecto, o Lucene é bastante eficiente. Sua linguagem de consulta é flexível e permite encontrar os itens com base em vários tipos de busca, combinando palavras-chave e operadores para criar consultas complexas e dinâmicas.

Para o desenvolvedor, essa eficiência é uma boa notícia. Contudo, o usuário do sistema não precisa saber disso. Se ele buscar por "pen drive da sandisk", o sistema deve entender o que o cliente está procurando, que neste caso é o produto de uma marca específica. O resultado dessa busca deve mostrar inicialmente os pen drives da sandisk e, em seguida, os pen drives de outras marcas ou outros produtos sandisk. O usuário vai focar nos resultados iniciais e encontrará o produto ideal para sua necessidade.

Talvez esse seja um exemplo simples e direto. No entanto, podemos imaginar algumas situações menos óbvias. O que dizer da consulta "dentista em santo andré"? Ora, você está procurando um dentista em Santo André. Novamente, os primeiros resultados mostrados no resultado da busca devem ser de dentistas cadastrados naquela cidade. E se eu pesquisar por "odontólogo em santo andré"? O sistema de busca precisa fazer mais um processamento para entregar o mesmo resultado, que será a realização da busca com os sinônimos de odontólogo.

E se eu mudar mais um pouco a pesquisa para "o melhor restaurante japonês de santo andré"? Neste caso, mais que o resultado da busca eu preciso de uma análise textual para descobrir de alguma forma a opinião das pessoas sobre os estabelecimentos. O nome dessa área é Análise de Sentimento, que está associada ao Processamento de Linguagem Natural, uma subárea da Inteligência Artificial.

Essa grande evolução na busca foi encabeçada pelos buscadores web, que nos permitem realizar consultas *ad hoc* em qualquer conteúdo textual. Na verdade, neste momento estamos em um nível um pouco adiante, onde as buscas são executadas em imagens e até em áudio e vídeo.

Fica claro que, mais do que apenas procurar por palavras-chave, precisamos entender que um sistema de busca tem de ser projetado (e pensado) para atender aos usuários menos especializados. É o que chamamos de encontrabilidade, que é a facilidade de encontrar uma informação específica dentro de um conjunto de documentos. Veja o comentário do Peter Morville, um grande especialista em arquitetura da informação:

> Encontrabilidade ambiental está mais relacionada às complexas interações entre humanos e informação que aos computadores. Tradução do autor para o trecho do livro *Ambient Findability*, de Peter Morville.

A qualidade do seu sistema de busca será, então, medida pela encontrabilidade das informações. Neste livro vamos tentar criar uma solução de busca que seja o mais próximo possível do usuário final a que ela se destina. Deve-se considerar todas as inovações que fazem parte hoje do cotidiano das pessoas, como o autocompletar e as sugestões. Para conseguirmos alcançar esse objetivo, teremos de usar todos os recursos disponíveis nas

ferramentas, criando soluções um pouco mais avançadas e complexas.

Agora que temos definido o que é um sistema de busca, precisamos entender quais tipos de dados serão utilizados. Há basicamente três formas a serem consideradas: dados estruturados, dados semiestruturados e dados não estruturados. Eles representam a fonte de dados primária para nossas aplicações de busca.

2.7 Dados estruturados

Dados estruturados são aqueles representados através de linhas e colunas, onde cada coluna tem um rótulo conhecido. Eles seguem um modelo predefinido e existe uma organização de alto nível, com restrições que delimitam a forma de inclusão de novos registros. Neste caso, a estruturação permite a um computador ler e interpretar a informação, que é o equivalente a dizer que um computador pode abrir e encontrar as informações dentro daquele modelo de dados já definido.

É o caso do esquema de um banco de dados relacional. Dizemos que esses dados são estruturados porque as colunas das tabelas são fixas. Você só pode inserir registros em uma coluna existente na tabela. Assim, o modelo de dados e suas restrições são conhecidos antes de inserir os valores.

Uma boa parte dos sistemas de informação é construída usando o modelo entidade-relacionamento, onde os dados são estruturados em esquemas, tabelas e colunas. Os valores estão organizados em blocos relacionados, por isso, podemos utilizar um banco de dados relacional. Dados de um mesmo grupo possuem os mesmos atributos e o mesmo formato são registros em uma tabela, listados sequencialmente. Não há surpresas.

Exemplo 1:

Imagine uma tabela fictícia chamada *Pessoa*. Ela poderia ter as colunas ID, nome, endereço e telefone. Todos os registros terão exatamente os mesmos campos, ainda que os valores sejam nulos. Se a coluna *nome* for definida com tamanho máximo de 50, não vai conseguir gravar 51 caracteres. Perceba que a coluna *endereço* (espera-se) tem sempre o endereço daquela pessoa. E o telefone (geralmente) será preenchido com uma cadeia de números. E você pode inserir um registro com *CPF*? Não, esse campo não está definido na estrutura da nossa tabela. Para poder usá-lo, você terá de mudar a estrutura antes da inserção.

Seguindo esse princípio da estruturação previamente definida, as consultas realizadas em bases relacionais costumam ser orientadas à coluna. Uma consulta à tabela *Pessoa* utilizando SQL seria algo parecido com isso:

```
SELECT ID, NOME, ENDERECO, TELEFONE FROM PESSOA
    WHERE NOME LIKE 'MARC%';
```

O resultado esperado é uma lista de pessoas com os nomes iniciando por *marc*, incluindo suas variações como *marco*, *marcia*, *marcos* e *marcone*. Dessa forma:

ID	NOME	ENDERECO	TELEFONE
2	MARCO REIS	RUA 37, BRASILIA	81194622
32	MARCOS PAULO	RUA 73, SAO PAULO	92567890
53	MARCONE PERILO	RUA 3333, RIO DE JANEIRO	98976789
71	MARCO SILVA	RUA 7777, SALVADOR	87876543
98	MARCIA LIMA	RUA DOS ALFENEIROS, 13	981194621

É o tipo de consulta que funciona bem para bases pequenas. As limitações, inclusive de performance, aparecem quando a base escala para milhões de registros. À medida em que a quantidade de registros aumenta, a velocidade da resposta diminui. Vale lembrar que uma base de dados muito grande se transforma em candidata para ser indexada com o Lucene.

Exemplo 2:

Agora, consideremos outra situação. Estamos trabalhando com um sistema de culinária e a principal tabela será *Receita*, onde são armazenados os dados de preparo para produtos alimentícios. Os campos dessa tabela são:

- ID (identificador numérico único)
- Nome (título da receita)
- Ingredientes (lista com os produtos necessários)
- Modo de preparo (descrição dos passos a seguir)
- Tempo de preparo (em minutos)
- Classificação (doce, salgada, azeda e amarga)

Cada nova receita precisa ter todos esses campos preenchidos com seus valores, de forma previsível. Não é permitido cadastrar uma receita sem os ingredientes ou modo de preparo. E quando sua base de receitas estiver completa, podemos fazer perguntas bem definidas e objetivas, como:

- Qual a receita mais demorada?
- Qual o tempo médio de preparo?
- Quais as receitas de alimentos doces?
- Combinação de vários campos: qual o tempo médio de preparo de uma receita amarga com nome de brigadeiro?

Esse tipo de questão é facilmente respondido com dados estruturados, porque o comportamento da tabela é previsível e determinado previamente, antes de se inserir os registros. O problema só começa a aparecer quando a complexidade das

questões aumenta e o usuário tem uma pergunta mais complicada. Por exemplo: qual receita de sobremesa não contém glúten?

2.8 Dados semiestruturados

Os dados semiestruturados são uma forma de dados estruturados, porém, não são dados no modelo relacional. Eles não são divididos em linhas e colunas, como uma tabela de banco de dados relacional. De uma forma diferente, os dados semiestruturados contêm marcadores que identificam e separam seus elementos. Os marcadores funcionam como rótulos, servindo como estruturas autodescritivas, pois contêm a definição do elemento e seu respectivo valor. Um marcador chamado de *telefone* contém informações telefônicas, assim como um marcador *CPF* deve conter o número de um CPF.

As formas mais comuns deste tipo de informação são XML e JSON. Enquanto os dados estruturados são representados por linhas e colunas com rótulos, os dados semiestruturados trazem os nomes dos campos junto a seus dados.

O XML é relativamente recente e representa uma nova maneira de representar e consultar dados. O modelo relacional data de 1970, enquanto a primeira versão do XML data de 1998. Dentre as motivações para sua criação, temos a necessidade de formatos mais flexíveis para representação de informação e a integração entre sistemas e plataformas.

Exemplo:

Mais uma vez, considere os dados de uma *Pessoa*, mas desta vez no formato XML e não no formato tabular visto anteriormente. A listagem a seguir mostra como seria sua representação. Perceba que são apenas os dados fictícios de duas pessoas.

```
<Pessoas>
    <Pessoa>
        <id>2</id>
        <nome>Marco Reis</nome>
        <endereco>Rua 37, BRASILIA</endereco>
        <telefone>81194623</telefone>
    </Pessoa>

    <Pessoa>
        <id>3</id>
        <nome>Diego Lucas</nome>
        <endereco>Rua 38, GOIANIA</endereco>
        <telefone>98798097</telefone>
        <cpf>8328676789</cpf>
    </Pessoa>
</Pessoas>
```

Este documento XML pode ser usado para transferir as informações entre sistemas distintos que usam diferentes linguagens de programação. E praticamente todas elas, Python, Ruby, Cobol, PHP etc., entendem e processam XML com facilidade. E este ponto deve ser anotado: dados estruturados e semiestruturados são facilmente processados por computadores.

O XML também tem a característica de ser flexível. Note que o segundo elemento, aquele com ID = 3, tem um atributo a mais (`cpf`). Os dados semiestruturados permitem esse tipo de construção. Os campos da pessoa com ID = 2 são `id` , `nome` , `endereco` e `telefone` . Os campos da pessoa com ID = 3 são `id` , `nome` , `endereco` , `telefone` e `cpf` .

Como a estrutura é diferente para as duas pessoas representadas no XML (uma tem CPF e a outra não), dizemos que esses dados são irregulares ou incompletos, ou seja, em alguns casos podem ter mais campos ou menos campos, de uma forma imprevisível.

No nosso exemplo isso ficou claro porque a estrutura tem campos diferentes, ou irregulares. O documento XML tem uma estrutura,

mas não há garantia de que ela seja fixa. Você pode ter mais ou menos campos.

Cada *tag* XML descreve o dado a que se refere. Mesmo sem uma documentação fornecida, entendemos que se trata dos dados de algumas pessoas. Alguns autores sugerem que XML e JSON são também dados estruturados. Mas aqui no livro vamos seguir a linha de pensamento que considera esses formatos como semiestruturados.

Apesar de ter muitas semelhanças, vamos considerar que a diferença fundamental entre os dados estruturados e semiestruturados é que os dados estruturados são tabulares, isto é, têm linhas e colunas como em um banco relacional, enquanto os dados semiestruturados são baseados em formatos como XML, JSON ou mesmo YAML.

2.9 Dados não estruturados

Dados não estruturados se referem a conteúdos textuais ou multimídia, onde o significado não é claro ou evidente e que é difícil de ser processado por um computador. Como exemplos podemos ter mensagens de e-mail, documentos do Word, vídeos, fotos, áudio e apresentações. Nesses casos, não é possível prever o que o usuário vai escrever. Por mais que tentemos criar um padrão, não há como validar se o texto segue essas regras.

Exemplo:

Nosso sistema de receitas é um bom modelo para observar o que seriam dados não estruturados. Agora, nossas receitas estão gravadas em arquivos de texto, como uma série de PDFs ou documentos do Word, ou seja, não estão em uma tabela organizada

em campos. Contudo, cada arquivo pode conter as mesmas informações do sistema com dados estruturados:

- ID (identificador numérico único);
- Nome (título da receita);
- Ingredientes (lista com os produtos necessários);
- Modo de preparo (descrição dos passos a seguir);
- Tempo de preparo (em minutos);
- Classificação (doce, salgada, azeda e amarga).

A diferença é que nesta nova situação o texto está escrito em linguagem natural, então temos texto puro em um arquivo do Word por exemplo. Assim, estamos tratando de *dados não estruturados*, aqueles que não têm estrutura fixa e já não é tão simples para um computador responder as mesmas questões levantadas anteriormente:

- Qual a receita mais demorada?
- Qual o tempo médio de preparo?
- Quais as receitas de alimentos doces?
- Combinação de vários campos: qual o tempo médio de preparo de uma receita amarga com nome de brigadeiro?

A maior parte das informações digitais do mundo está em formato não estruturado, ou seja, texto. Os dados estruturados e semiestruturados respondem por cerca de 10% a 20% e os dados não estruturados ficam com os 80% restantes. É uma quantidade imensa de conteúdo que não para de crescer. Considere a internet como uma fonte de dados não estruturados em expansão. Tendo isso em mente, empresas como Google, Bing, Facebook e Twitter, só para citar os mais famosos, estão desenvolvendo ferramentas para analisar e extrair valor dessa fonte de dados. E ganhar muito dinheiro, claro.

Neste livro, a ideia é misturar todos os tipos de dados, estruturados ou não estruturados, exatamente como acontece nas demandas por sistemas corporativos. Vamos trabalhar com documentos do Word,

PDF, registros de banco de dados, XML, JSON e logs, com o objetivo de cobrir muitas possibilidades de cenários reais. Essa mistura é o que mais nos interessa e que representa o maior desafio para os sistemas de busca. A interpretação de textos não é uma atividade simples nem para as pessoas, quanto mais para computadores.

2.10 Índice invertido

O Lucene realiza buscas e encontra documentos rapidamente, mesmo quando tem de analisar bilhões de documentos. Como isso é possível? Na verdade, não existe mágica e sim matemática. Os sistemas de busca, incluído aqui o Lucene, armazenam os dados em uma estrutura chamada de *índice invertido*. Esse nome é adequado ao seu princípio de funcionamento, que é similar ao índice de um livro. Em um livro, o índice é o mecanismo usado para encontrar rapidamente a página onde se encontra um determinado termo.

Ora, se você estiver lendo um livro de Direito Constitucional e precisa encontrar as páginas que contêm o termo *habeas corpus*, o que faria? Se não tiver o índice disponível, você teria de ler todas as páginas até encontrar as que contêm os termos procurados. A título de curiosidade, esse tipo de obra costuma ter em torno de 1000 páginas.

Com o índice tudo fica mais fácil. Ele contém os termos e as páginas em que ocorrem. Dessa forma, consultando o índice do livro de Direito Constitucional, descobrimos as páginas onde aparece o termo *habeas corpus*. Sabendo quais as páginas onde existe o termo pesquisado (*habeas corpus*) podemos abrir o livro e consultar apenas esses trechos. Percebe que é mais rápido acessar um índice do que ler todo o livro? A limitação de um índice literário é evidente. Não é possível ter todos os termos indexados, serão

teríamos outro livro só com o índice, dessa forma, o mais comum é que o índice contenha apenas os termos mais importantes.

Por esse motivo chamamos de *índice invertido*. O buscador não precisa varrer todos os documentos à procura dos termos que o usuário quer encontrar. É exatamente o inverso disso. O buscador acessa o índice e encontra os termos pesquisados. Como cada termo está associado aos documentos onde ocorre, com pouco processamento o buscador consegue devolver a lista de documentos de que o usuário precisa. Cada termo aponta para uma lista de documentos. Essa estrutura é projetada para permitir busca textual com grande velocidade e baixo custo de processamento. Vamos analisar algumas situações extremas nos exemplos adiante.

O conceito de índice invertido é muito importante se você pretende fazer otimizações no seu sistema de busca. Saber como os dados são organizados vai ser útil quando precisar criar soluções complexas. E um índice invertido é, antes de qualquer coisa, um índice. Ele segue mais ou menos o mesmo caminho que um índice para ser criado. Veja:

1. Coletar os documentos;
2. Separar o texto em palavras (*tokens*);
3. Processamento linguístico dos termos;
4. Indexar os documentos onde existem as ocorrências do termo.

Ok, não é exatamente simples. Vejamos o exemplo a seguir para esclarecer. Depois de um tempo fica bem simples de entender. Consideremos uma coleção de 3 documentos, cujo conteúdo você pode conferir logo a seguir.

ID	Conteúdo
Documento 1	java é uma linguagem de programação
Documento 2	o estudo da linguagem é chamado de linguística

ID	Conteúdo
Documento 3	o roteiro da programação musical do fim de semana terá shows de rock

As palavras de baixa relevância (é, uma, de, da, do, o) de um texto são chamadas de *stop words* e devem ser retiradas antes da indexação (veremos mais logo em seguida). O Lucene já tem uma lista de *stop words* em diversas línguas, incluindo português. O índice invertido para essa coleção ficaria assim:

Termo	Documento
fim	`[3]`
java	`[1]`
linguagem	`[1, 2]`
linguística	`[2]`
musical	`[3]`
programação	`[1, 3]`
roteiro	`[3]`
rock	`[3]`
semana	`[3]`
shows	`[3]`

Analisando a tabela, verificamos que a palavra *java* só existe no documento 1, enquanto *linguagem* está no documento 1 e no documento 2. Continuando, o termo seguinte é *programação* e ocorre nos documentos 1 e 3. Agora ficou claro porque chamamos de índice invertido. A busca, de fato, não é feita nos *documentos* e sim na lista de termos extraídos de cada documento. Por isso é que

a busca com Lucene é tão rápida. A lista de termos é que aponta para os documentos.

Sem dúvida, varrer os documentos é mais demorado que simplesmente acessar o índice invertido com a lista de termos e suas ocorrências. Vamos pensar como o Lucene neste ponto. Considere a tabela com o índice invertido. Quando o usuário pesquisar pela palavra *linguagem*, o Lucene vai acessar a tabela e encontrará rapidamente as ocorrências para o termo. Veja que *linguagem* é o segundo termo da nossa tabela e ocorre nos documentos 1 e 2. Isso foi feito sem precisar abrir esses documentos e sem gastar mais recursos como processamento ou memória.

Exemplo 1: você quer consultar os documentos com a palavra *zumbi*. Seu índice invertido criado pelo Lucene está ordenado alfabeticamente, então, essa consulta vai precisar apenas das palavras que começam com a letra *Z*, lá no final do índice. Com isso, o Lucene ignora praticamente todo o índice e vai direto para o final. Com pouco processamento ele encontra os documentos que estão associados à palavra *zumbi*.

Exemplo 2: a consulta agora é com a palavra *abarcar*, logo é uma palavra que fica bem no começo do índice. Com isso, depois de encontrar o que procura, o nosso Lucene pode retornar o resultado para o usuário com a certeza de que entregou tudo o que era necessário.

Stop words

As *stop words* são palavras de pouca importância dentro de um texto, como preposições e artigos. Não faz sentido indexar essas palavras porque elas não adicionam significado aos termos. Quando você precisa fazer uma busca por notícias de futebol, pode

pesquisar por *seleção de futebol*. Nesse caso, existe uma *stop word* que não acrescenta significado, a preposição *de*. Seria o mesmo que pesquisar por *seleção futebol*, seguindo a ideia de retirar as palavras sem importância.

Exemplo 1: o usuário quer saber as novidades da área de economia. Então ele acessa seu site preferido e procura por *notícias de economia*. A preposição *de*, neste contexto, é uma *stop word*, ou seja, totalmente irrelevante porque qualquer documento tem várias ocorrências dessa palavra. Imagine que se você pesquisar em um site qualquer por notícias com a palavra *de*. Ora, isso vai recuperar todo o conteúdo do site, sem nenhum filtro, uma vez que praticamente todos os textos em português têm ao menos uma palavra *de*. É o tipo de consulta que não tem sentido. É como acessar o site de um jornal e pesquisar por *notícias*.

A retirada das *stop words* também pode trazer problemas, mas o percentual de erros é pequeno se comparado ao benefício. A quantidade de termos que precisam mesmo das *stop words* é pequena. Nesses casos, a perda de precisão é aceitável. Algumas situações precisam realmente das *stop words*, como os substantivos compostos ou nomes próprios. Exemplos: *A favorita*, *Guerra dos sexos*, *Cordilheira dos Andes* e *Organização das Nações Unidas*.

Relevância

O resultado da busca deve ser classificado de acordo com a relevância de cada documento, garantindo que os itens relevantes para a consulta do usuário apareçam primeiro na lista. Tomando como base o site de e-commerce, quando o usuário pesquisa por *pen drive sandisk*, o resultado deve mostrar os produtos mais importantes (ou relevantes) no começo. Dessa forma, podemos dizer que a *relevância* define o grau de importância de um documento dentro do conjunto de documentos pesquisados, que é o

índice. Então, a relevância diz se um documento é ou não interessante para uma consulta específica do usuário. O cálculo da relevância do documento é usado para ordenar os itens no resultado da consulta, dessa forma, o primeiro item do resultado da busca é o mais relevante e os demais aparecem em ordem decrescente.

Esse cálculo varia de acordo com o tipo de informação que estamos processando. Um sistema com dados médicos precisaria de algoritmos diferentes de um sistema com documentos jurídicos. Nesses casos, a relevância é calculada de forma diferente. O Lucene possui algoritmos genéricos para essa classificação. Quando nós pesquisamos por *java*, o motor de busca vai retornar itens como *Ilha de Java* e *linguagem de programação java*. De uma forma geral, quanto mais vezes o termo aparece no documento, mais relevante ele é. Inversamente, quanto mais vezes o termo ocorre através dos documentos de uma coleção, menos relevante ele se torna. Essa é a ideia por trás do TF-IDF.

A equação usada para calcular a relevância de forma genérica é chamada de TF-IDF ou *Term Frequency/Inverse Document Frequency*. Entender seu funcionamento nos permite criar buscas melhores. A base do Lucene é o TF-IDF, mas há outros algoritmos diferentes. Por ser amplamente usado, vamos focar no TF-IDF. Como o nome sugere, o cálculo é feito em duas fases: TF e depois IDF. Os dois valores formam uma medida de ponderação que funciona bem na maioria dos casos, daí sua popularidade.

O *TF* (Term Frequency) representa a frequência com que o termo aparece no documento. Quanto mais vezes o termo aparece no documento, maior o seu peso. Um campo que contém várias menções a uma palavra (ou termo) é mais *provável* de ser relevante que um campo que contém apenas uma menção. Não é essencial você saber calcular o TF-IDF, o Lucene faz isso automaticamente. No entanto, mal não faz saber como é o funcionamento interno da ferramenta.

O TF é calculado assim:

- TF(termo no documento) = vfrequência de vezes que o termo ocorre no documento;
- O TF de um termo no documento é a raiz quadrada da quantidade de vezes que ele ocorre no documento.

O *IDF* (*Inverse Document Frequency*) representa a importância do termo na coleção de documentos. O TF considera que todos os termos são igualmente importantes, mas na prática isso não é verdade. Palavras com muitas ocorrências na coleção de documentos tendem a ser menos importantes. É o caso, por exemplo, das preposições. Uma lista não exaustiva seria: a, ante, após, até, com, contra, de, desde, em, entre, para, per, perante, por, sem, sob, sobre, trás. Essas palavras não ajudam no resultado de sua busca, porque não são importantes na busca por itens de uma coleção de documentos. Mesmo que apareçam muitas vezes, sua relevância é baixa no significado das frases. Isso implica que termos menos comuns são mais relevantes que termos muito repetidos nos documentos. A equação do IDF é:

- IDF(termo) = 1 + log (quantidade de documentos / (docFreq + 1));
- Onde: docFreq é o número de documentos onde o termo ocorre.

Temos mais algumas variáveis para classificar a relevância do documento. Uma delas é o *fieldNorms*, que reflete o tamanho do documento. Quando uma palavra aparece em um documento pequeno, sua relevância é diferente de quando essa palavra aparece em um documento extenso. Os termos são mais concentrados em documentos pequenos, então, é mais provável que esse documento tenha uma relevância maior. Ok, talvez não seja a melhor forma de calcular a relevância, mas é uma fórmula genérica que funciona razoavelmente bem.

Em função dessa imprecisão, o Lucene, a partir de sua versão 6, mudou um pouco o cálculo da relevância. Agora o algoritmo padrão é o BM25, uma evolução do TF-IDF. O BM25, acrônimo de *Best*

Match 25, foi lançado em 1994 na *Third Text REtrieval Conference (TREC 1994)* (http://trec.nist.gov/pubs/trec3/t3_proceedings.html/) e é baseado em equações de probabilidade, isto é, a relevância é, agora, a probabilidade de o usuário considerar aquele resultado relevante.

Precisão e revocação

Medir a relevância do resultado de uma consulta é uma tarefa complicada. Para nos ajudar a resolver esse problema temos a precisão e revocação. Precisão e revocação são conceitos centrais para calcular a relevância do resultado de uma consulta. Quando o sistema recupera documentos relevantes, podemos julgar que ele é eficiente em sua funcionalidade, logo, um bom motor de busca recupera a maior quantidade possível de itens relevantes, com a menor quantidade possível de itens irrelevantes. Essa é a ideia da precisão e revocação.

Seu cálculo é dado pela fórmula:

Precisão = quantidade de documentos relevantes recuperados / quantidade de documentos retornados pela busca

A *precisão* avalia se os documentos recuperados pela consulta estão limpos de ruídos. Vamos pensar novamente na consulta por *pen drive*. Podemos considerar que o esperado pelo usuário é uma lista com os *pen drives* disponíveis para venda no nosso e-commerce. Mas um dos produtos retornados pode ser, por exemplo, um computador que tenha suporte a *pen drive*. Ora, se o usuário esperava apenas *pen drives*, um computador é um ruído no resultado.

Revocação = quantidade de documentos relevantes recuperados / quantidade de documentos relevantes da coleção

A *revocação* demonstra a abrangência do resultado da consulta. O cálculo da equação está entre zero e um, sendo que quanto mais próximo de um, melhor será a revocação.

Imagine agora uma consulta (A) qualquer que retornasse todos os documentos da sua coleção (C). Significa que A = 100 e C = 100, ou seja, sua consulta retornou 100 documentos e sua coleção tem 100 documentos. Aplicando os valores na equação temos que:

Revocação = 100 / 100

O resultado da equação nesse caso é um, o que representa um ótimo índice de revocação. A revocação compreende outro conceito importante, que é a *cobertura*, ou a abrangência do resultado da consulta.

Então surge outra dúvida. Como determinar o que é um documento relevante? O professor Lancaster, em seu artigo *The measurement and evaluation of library services*, de 1977, sugeriu algumas possibilidades. A mais utilizada é comparar uma consulta real (A) do usuário com consultas semelhantes (B, C, D...) de outras pessoas. Depois, solicita-se aos usuários que eles indiquem os itens relevantes.

É uma técnica muito usada em e-commerce. Quando os clientes buscam por um termo específico, por exemplo *pen drive*, os produtos mais relevantes serão aqueles com mais cliques. Assim, podemos treinar nosso sistema de buscas para mostrar os itens mais clicados para aquela consulta do usuário. Pensando na interface do usuário, poderíamos mostrar uma mensagem indicando quais os produtos mais indicados na categoria *pen drive*, algo como uma lista de produtos semelhantes.

Com isso, podemos dizer que a revocação é a capacidade do sistema de recuperar documentos úteis para o usuário. Do outro lado, a precisão é a capacidade de evitar documentos inúteis. Uma

curiosidade é que as duas medidas trabalham de forma inversa. Quando melhoramos a precisão, piora a revocação. E o inverso é também verdadeiro: quando melhora a revocação, piora a precisão. Isso impõe um limite ao melhor resultado que podemos oferecer ao usuário. No entanto, há técnicas que nos permitem ir além desses limites.

O aumento da precisão passa pela exatidão dos descritores. Em nossa consulta por *pen drive*, para melhorar a precisão adicionamos mais um termo para melhorar a descrição do que se pretende buscar. Nossa busca deveria ser mais específica, como *dispositivo de armazenamento pen drive*.

Outro exemplo interessante seria a busca por *oscar* em um portal de notícias. O resultado da consulta traria itens sobre o jogador de basquete e sobre a premiação do cinema. Para resolver o problema da precisão, o motor de busca deve especializar a consulta, sugerindo ao usuário as duas opções: o jogador ou o prêmio. Esse é o problema dos homônimos, termos que têm mais de um significado.

Para melhorar a precisão, temos essas técnicas:

- Distinção de homônimos: diferenciar entre Java (a ilha) e Java (a linguagem de programação da Oracle);
- Peso de um determinado termo: se o termo pesquisado está no título, seu peso tende a ser maior;
- Ligação entre os documentos: verificar se há algum tipo de ligação entre o conteúdo dos documentos.

E para melhorar a revocação podemos utilizar:

- Controle de sinônimos: cria-se um dicionário de sinônimos com vocabulário controlado, então, quando indexar a palavra *bonito*, adicionamos a palavra *belo* e *lindo* ao mesmo documento;
- Agrupamento das formas das palavras, com o radical, singular e plural: durante a indexação, adicionar as outras formas da palavra. Exemplo: andarilho, andarilhos, andarilha e andar;

- Agrupamento de conceitos semelhantes ou relacionados: uma busca por *star trek* pode retornar documentos sobre *capitão kirk* ou *senhor spock*, que são conceitos relacionados;
- Ligação hierárquica: um documento pode ter hierarquia, como no Direito Criminal onde existe a figura do *Pedido de Prisão* e subclasses como *Pedido de Prisão Temporária* e *Pedido de Prisão Preventiva*.

Opcionalmente, o controle de sinônimos pode ser aplicado aos quase sinônimos, como em *feliz*/*alegre* ou *córrego*/*riacho*, que são vocábulos próximos, no entanto, não são idênticos.

Os conceitos não são muito fáceis de entender, porém precisam ser estudados para melhorar a qualidade do motor de busca.

Resumo

Os sistemas de busca evoluíram de acordo com a necessidade de informação dos usuários. Suas funcionalidades, hoje, vão além da simples busca por palavra-chave ou pelo código utilizando a linguagem SQL de um banco de dados relacional. Para desenvolver um trabalho completo e aproveitar ao máximo os recursos do Lucene é essencial entender os conceitos em torno do assunto da Recuperação de Informação, como documento, campo, dados estruturados/semiestruturados/não estruturados.

Claro que alguns conceitos são menos utilizados. Os capítulos iniciais usam apenas os conceitos básicos para aprendermos a sintaxe básica do Lucene, até porque um motor de busca mais simples não precisa considerar questões como relevância, precisão e revocação. Contudo, esses conceitos são significativos em motores de busca especializados, como os que lidam com dados médicos ou jurídicos.

Deste capítulo de conceitos, o conhecimento mais importante é o *índice invertido*, que você deve manter em mente pois explica em grande parte como funciona o Lucene e sua performance. Um índice

invertido é utilizado para agilizar as buscas. É uma técnica muito eficiente para este tipo de situação é tende a ser mais performática que uma consulta similar em um banco de dados relacional. O SGBDR precisa acessar todos os seus registros e, quando lidamos com bases grandes, isso é lento.

O índice invertido não precisa acessar todos os documentos indexados, pelo contrário. Ele acessa diretamente as palavras que o usuário está pesquisando e retorna os documentos que estão associados a elas. Por isso, é possível encontrar uma determinada palavra (ou conjunto de palavras) rapidamente dentro de uma grande coleção de textos. Contudo, há um custo grande para gerar o índice porque os textos precisam ser pré-processados antes de estarem disponíveis para consulta.

O TF-IDF é uma medida genérica para calcular a relevância de cada item no resultado de uma consulta do usuário. Ele leva em consideração a consulta do usuário, a quantidade de palavras de cada documento e a quantidade de documentos do seu índice. Não é necessário entrar em detalhes sobre esse assunto, a menos que você pretenda implementar uma forma personalizada de ordenar o resultado de suas buscas.

Veja o caso do Google e Bing. Eles têm métodos personalizados que levam em consideração o seu histórico de buscas e perfil pessoal. Imagine duas pessoas (um programador e um viajante) e a mesma consulta (java). O Google vai mostrar resultados referentes à linguagem Java para o programador e, para o viajante, vai mostrar um resultado centrado na ilha de Java, aquela que fica na Indonésia.

Até agora ficamos no campo teórico. No próximo capítulo veremos na prática, através de um projeto Java, como implementar o nosso primeiro sistema de buscas. Aplicaremos os conceitos e usaremos os termos técnicos discutidos aqui. Neste projeto criaremos um buscador para os arquivos locais do seu computador que pode ser

facilmente estendido para outras fontes de dados, como diretórios compartilhados na rede.

Capítulo 3
Indexação e busca

O objetivo deste capítulo é mostrar o processo de indexação e busca com o Lucene. Primeiramente, vamos indexar os arquivos e em seguida vamos recuperá-los através de buscas com palavras-chave. Escolha um diretório no seu computador que contenha alguns arquivos pessoais. Pode ser arquivos do Word, PDFs, RTF etc. O importante é ter uma boa quantidade de arquivos para verificar o funcionamento do Lucene.

É possível traçar um paralelo com as bibliotecas tradicionais de livros. Se antigamente você precisava ir à biblioteca da sua cidade para pesquisar um assunto, agora, com um sistema de buscas como esse, buscamos as informações em sua biblioteca de arquivos pessoais, com resultados mais abrangentes e completos.

Para encontrar esses documentos, o tipo mais simples de busca é por palavra-chave, segundo o qual o Lucene retornará os itens que contêm ao menos uma vez o termo informado. Por exemplo buscando pelo termo *java*, a aplicação vai retornar os itens que têm essa palavra em algum lugar do texto. Esse é o primeiro passo: encontrar os documentos que atendem ao critério de busca. O segundo passo é ordenar esses documentos de acordo com a sua importância. O documento mais importante vem primeiro e assim por diante. Essa ordem é definida pelos cálculos de relevância, discutidos no capítulo anterior.

3.1 O que vamos precisar

Os exemplos deste capítulo (e do resto do livro) foram escritos no Eclipse com Java 8, mas funcionam em qualquer IDE. O código está

disponível no GitHub neste link: https://github.com/masreis/exemplos-livro-lucene/. A partir do Lucene 6 é necessário usar o Java 8.

Mas não se preocupe com todos os detalhes, parâmetros e opções, afinal, este será apenas o primeiro exemplo. Em capítulos seguintes haverá mais explicações, seguindo uma ordem crescente de complexidade. Concentre-se na funcionalidade, ao menos nesse primeiro momento.

O Lucene indexa apenas texto, no entanto, os arquivos PDF e DOCx do seu computador são binários, ou seja, não contêm texto puro. Para extrair o conteúdo textual desses arquivos precisamos de uma ferramenta específica, o Apache Tika. Esta é uma biblioteca que detecta e extrai o texto de diversos tipos de arquivo.

O *parser* do Tika encapsula toda a complexidade do processamento de diversos tipos de arquivos e extrai o conteúdo de forma transparente. É uma ferramenta muito interessante, mas falha em alguns casos, particularmente em PDFs com formatação complexa. Quando não consegue analisar o arquivo, ele vai apenas lançar uma exceção e não extrairá aquele conteúdo específico.

A versão do Tika utilizada neste livro suporta os seguintes formatos: HTML, XML, Microsoft Office, ODF (OpenOffice), PDF, ePUB, RTF, TXT, arquivos compactados, arquivos de áudio/imagem/vídeo (apenas metadados) e MBOX (correio eletrônico).

Com o Tika podemos fazer buscas no conteúdo de arquivos desses tipos, que representam a grande maioria das opções disponíveis. Além do Tika, nossos projetos usam Apache Maven, Log4j, JUnit, MySQL, JSF e JPA, quer dizer, é uma típica aplicação corporativa escrita com a plataforma Java.

Nossa primeira aplicação é um motor de busca para os arquivos locais na sua máquina. Se você usa Windows ou Mac, de antemão, deve estar acostumado com o *Windows Search* ou *Spotlight*, que são as respectivas ferramentas de busca. Com elas, o usuário pode

facilmente encontrar documentos no disco com através de consultas por palavra-chave. Exemplo: encontrar os arquivos com extensão *PDF*, arquivos modificados essa semana ou arquivos que contêm a palavra *java* em seu conteúdo.

3.2 Primeira fase: indexação

Como visto no capítulo sobre conceitos, um sistema de busca tem 2 módulos diferentes: indexador e buscador. A indexação torna os documentos encontráveis através de uma consulta. Nosso programa vai ler um diretório, que deve conter a lista de arquivos para indexação, ou a nossa coleção de documentos. Os arquivos devem estar em um dos formatos suportados pelo Tika. O módulo indexador vai analisar o conteúdo de cada arquivo e gerará o índice ao final do processamento.

O diretório do índice será usado pelo Lucene para gravar o resultado da indexação. Minha sugestão é usar `/livro-lucene/indice`. Este diretório será acessado apenas pelo Lucene. O diretório de documentos é qualquer um que contenha arquivos com texto. Pode ser um diretório com documentos do Word, PDFs ou qualquer formato suportado pelo Tika. Por exemplo: você tem um diretório com seus arquivos pessoais, apostilas e livros. Quando quiser encontrar o livro que fala sobre *Java e Orientação a Objetos*, em vez de usar o sistema operacional você pode usar nosso sistema de buscas. A vantagem é que o nosso pode ser adaptado para outras situações, como diretórios da rede.

Com o diretório de documentos selecionado, é hora de indexar cada um dos arquivos. O Lucene, para relembrar, trabalha apenas com texto, o que significa que devemos extrair o conteúdo de arquivos binários (PDF, DOC, XLS etc.) com o Apache Tika. Depois, criaremos um documento no Lucene para cada arquivo local do seu

computador. Esse documento será adicionado ao índice e estará disponível para consultas.

Os atributos dos arquivos que vamos utilizar são:

- `conteudo` : texto do arquivo. No caso de documentos de texto como DOC e PDF, o Tika extrai o conteúdo textual. No caso de arquivos como MP3, extrai metadados básicos como nome da música, estilo, artista etc.;
- `tamanho` : a quantidade de bytes do arquivo;
- `data` : data de indexação do arquivo. É altamente recomendável a utilização da data em formato AnoMêsDia, ou *yyyyMMdd*, para a posterior recuperação desses documentos através de buscas por intervalo;
- `caminho` : nome completo com o seu caminho, para posterior recuperação do arquivo original.

A primeira fase da nossa aplicação de recuperação da informação terá o seguinte roteiro:

1. definir o diretório do índice Lucene;
2. criar um objeto `IndexWriter` ;
3. extrair o conteúdo de cada um dos arquivos que pretende indexar;
4. criar um documento Lucene com os dados do arquivo;
5. adicionar o documento no índice;
6. fechar o `IndexWriter` .

A classe IndexadorArquivosLocais

A primeira parte do nosso sistema de buscas é realizada pela classe `IndexadorArquivosLocais` . Ela recuperará os arquivos do disco e criará o índice, conforme descrito anteriormente. Está dividida em blocos para auxiliar no entendimento, sendo que os principais métodos são `inicializar` , `indexar` e `finalizar` . O código completo, se quiser dar uma conferida, está disponível no GitHub no projeto *exemplos-livro-lucene*.

Antes de executar o primeiro exemplo, selecione um diretório que contém poucos arquivos, por volta de 500, não mais que isso. Muitos arquivos pode ser um problema porque vai demorar bastante. É até razoável indexar o computador inteiro mais tarde, você apenas precisa ter tempo e paciência.

Os projetos deste livro usam Apache Maven, uma ferramenta para gerenciamento de projetos que facilita o *build* de aplicações Java, entre outras funcionalidades bem legais. Ele é baseado na ideia de *project object model*, ou apenas *pom*. A versão do Lucene é indicada pela *tag* `<lucene.version>` . Essas configurações ficam no arquivo `pom.xml` e podemos conferir uma parte do código logo a seguir:

```
    <properties>
        <version.compiler.plugin>2.3.1</version.compiler.plugin>
        <maven.compiler.target>1.8</maven.compiler.target>
        <maven.compiler.source>1.8</maven.compiler.source>
<lucene.version>7.4.0</lucene.version>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.apache.lucene</groupId>
            <artifactId>lucene-core</artifactId>
            <version>${lucene.version}</version>
        </dependency>

        <dependency>
            <groupId>org.apache.lucene</groupId>
            <artifactId>lucene-queries</artifactId>
            <version>${lucene.version}</version>
        </dependency>

        <dependency>
            <groupId>org.apache.lucene</groupId>
            <artifactId>lucene-analyzers-common</artifactId>
            <version>${lucene.version}</version>
```

```
        </dependency>

        <dependency>
            <groupId>org.apache.lucene</groupId>
            <artifactId>lucene-queryparser</artifactId>
            <version>${lucene.version}</version>
        </dependency>
    </dependencies>
```

Após a criação do arquivo *pom* podemos passar para o código da classe de indexação, com os atributos e métodos de acesso. Na continuação do capítulo veremos os detalhes da implementação do Lucene. O código parcial é o que segue:

```
public class IndexadorArquivosLocais {
    private static final Logger logger = Logger
            .getLogger(IndexadorArquivosLocais.class);
    private IndexWriter writer;
    private Directory diretorio;
    private Tika extrator = new Tika();
    private boolean recursivo;
    private String diretorioIndice;
    private String diretorioDocumentos;
    private long totalArquivosIndexados;
    private long totalBytesIndexados;
    private boolean apagarIndice;

    public void inicializar() throws IOException {
        if (apagarIndice) {
            FileUtils.deleteDirectory(
                    new File(diretorioIndice));
        }
        Analyzer analyzer = new StandardAnalyzer();
        diretorio = FSDirectory
                .open(Paths.get((diretorioIndice)));
        IndexWriterConfig conf = new IndexWriterConfig(
                analyzer);
        writer = new IndexWriter(diretorio, conf);
    }

    public void finalizar() {
```

```
            try {
                writer.close();
                diretorio.close();
                //
                logger.info("Total de arquivos indexados: "
                        + totalArquivosIndexados);
                logger.info("Total de bytes indexados (MB): "
                        + totalBytesIndexados / (1024 * 1024));
            } catch (IOException e) {
                logger.error(e);
            }
        }

        public void setApagarIndice(boolean apagarIndice) {
            this.apagarIndice = apagarIndice;
        }

        public void setDiretorioIndice(String diretorioIndice) {
            this.diretorioIndice = diretorioIndice;
        }

        public void setRecursivo(boolean recursivo) {
            this.recursivo = recursivo;
        }

        public void setDiretorioDocumentos(
                String diretorioDocumentos) {
            this.diretorioDocumentos = diretorioDocumentos;
        }
        //{...}
}
```

O começo da classe é responsável por definir as variáveis e diretórios do índice e dos documentos que vamos indexar. Veja que temos o IndexWriter e o Tika, além de outros detalhes sem relação com o Lucene. Os atributos e construtores estão definidos para facilitar a reutilização da classe em outras situações e não apenas neste exemplo. É o caso do atributo `recursivo`, que indica a recursividade em relação aos diretórios que serão indexados. Como

está definida para `false`, vai indexar apenas o diretório indicado. Se for alterada para `true` indexará o conteúdo dos subdiretórios.

Continuando o estudo da classe, no bloco seguinte temos o método `inicializar()`:

```
    public void inicializar() throws IOException {
        if (apagarIndice) {
            FileUtils.deleteDirectory(
                    new File(diretorioIndice));
        }
        Analyzer analyzer = new StandardAnalyzer();
        diretorio = FSDirectory
                .open(Paths.get((diretorioIndice)));
        IndexWriterConfig conf = new IndexWriterConfig(
                analyzer);
        writer = new IndexWriter(diretorio, conf);
    }
```

Ele vai criar o diretório do índice com a configuração mínima necessária e sem nenhuma otimização. Um novo índice é criado para cada execução e o diretório anterior é excluído com a instrução `FileUtils.deleteDirectory(File)`. Tenha isso em mente quando executar esse programa e se não quiser apagar um índice já criado, use outro diretório.

No método de inicialização, estamos criando os principais objetos do Lucene. Neste momento, não é necessário nenhum conhecimento mais profundo sobre eles. Podemos dizer que esses são os valores padronizados.

- `Analyzer`: faz o pré-processamento, transformando as palavras em letras minúsculas.
- `Directory`: representa o diretório do índice.
- `IndexWriterConfig`: configurações e otimizações. Usaremos por enquanto apenas os padrões.
- `IndexWriter`: responsável por criar o índice com a inclusão dos documentos.

A implementação do `Analyzer` utilizada neste primeiro programa é o `StandardAnalyzer`, um analisador genérico para textos. Ele faz a análise básica, convertendo o conteúdo para letras minúsculas, separando as palavras e removendo caracteres especiais. Na maioria dos casos é mais do que suficiente.

O método `indexar()` dá início ao processamento dos arquivos a partir do diretório raiz indicado pelo programador e chama o método de indexação para os arquivos gravados no computador.

```
public void indexar()
        throws IOException, TikaException {
    indexarDiretorio(new File(diretorioDocumentos));
}
```

O bloco do programa que efetivamente indexa os arquivos é o método `indexarArquivo(arquivo)`. Lembre-se de que cada arquivo será convertido em um objeto Document do Lucene. Neste exemplo utilizamos os tipos de dados `TextField`, `StringField` e `LongPoint`. Os dois primeiros são os tipos mais comuns do Lucene e representam o conteúdo textual, assim como uma *String* no Java. O `TextField` é usado para indexar textos longos. O `StringField` é indicado para campos exatos como datas e códigos. Nos dois casos, é possível realizar buscas nestes campos e armazenar o seu valor no índice. O `LongPoint` é mais recente e está disponível a partir da versão 6 do Lucene. É usado para indexar números e fazer buscas por intervalo de valores. Ele tem uma limitação, que é o fato de não armazenar o valor indexado, assim, se você precisa mostrar esse valor na tela do usuário, precisa armazená-lo em um campo separado, através de um `TextField` ou `StringField`. Logo, um campo `LongPoint` (ou `BinaryPoint`, `DoublePoint`, `FloatPoint` e `IntPoint`) é pesquisável, mas não armazena valores.

Neste bloco de código, aparece pela primeira vez a classe `Document`, a unidade básica de indexação do Lucene. Ela merece um pouco de atenção. Quando precisar indexar um registro de banco de dados ou

arquivo binário, deve-se utilizar a estrutura de um `Document` para memorizar as informações. Essa classe é, basicamente, um mapa com conjunto de campos chave-valor. Para cada chave está associado um ou vários valores e são referentes ao tipo de informação que queremos indexar. Vejamos um modelo para esclarecer. Como estamos indexando um arquivo do computador, nossos campos são:

- `conteudo` : o texto extraído do arquivo;
- `tamanho` : quantidade de bytes do arquivo;
- `tamanhoLong` : quantidade de bytes do arquivo em formato Long. Será utilizado no próximo capítulo para buscas por intervalo numérico;
- `data` : data de modificação do arquivo formatada utilizando o utilitário *DateTools* do Lucene;
- `caminho` : nome do arquivo, incluindo seu caminho completo;
- `extensao` : extensão do arquivo (PDF, DOC, XLS etc.).

```
public void indexarArquivo(File arquivo) {
    try {
        Document doc = new Document();
        Date dataModificacao = new Date(
                arquivo.lastModified());
        String dataParaIndexacao = DateTools
                .dateToString(dataModificacao,
                        Resolution.DAY);
        String extensao = consultarExtensaoArquivo(
                arquivo.getName());
        String textoArquivo = extrator.parseToString(
                new FileInputStream(arquivo));
        doc.add(new TextField("conteudo", textoArquivo,
                Store.YES));
        doc.add(new TextField("tamanho",
                String.valueOf(arquivo.length()),
                Store.YES));
        doc.add(new LongPoint("tamanhoLong",
                arquivo.length()));
        doc.add(new StringField("data",
                dataParaIndexacao, Store.YES));
```

```
            doc.add(new StringField("caminho",
                    arquivo.getAbsolutePath(), Store.YES));
            doc.add(new StringField("extensao", extensao,
                    Store.YES));
            writer.addDocument(doc);
            logger.info("Arquivo indexado ("
                    + (arquivo.length() / 1024) + " kb): "
                    + arquivo);
            totalArquivosIndexados++;
            totalBytesIndexados += arquivo.length();
        } catch (Exception e) {
            logger.error(
                    "Não foi possível processar o arquivo "
                            + arquivo.getAbsolutePath());
            logger.error(e);
        }
    }
```

O método `consultarExtensaoArquivo` é usado unicamente para retornar o formato do arquivo, separando as últimas letras do nome do arquivo. Em um arquivo chamado `teste-lucene.docx`, a extensão será `docx`. Isso vai permitir que executemos buscas de um tipo específico de arquivo, através de sua extensão.

```
    private String consultarExtensaoArquivo(String nome) {
        int posicaoDoPonto = nome.lastIndexOf('.');
        if (posicaoDoPonto > 1) {
            return nome.substring(posicaoDoPonto + 1,
                    nome.length()).toLowerCase();
        }
        return "";
    }
```

Após adicionar os documentos no `IndexWriter` devemos usar o método `finalizar()` para liberar os recursos utilizados e gravar o índice. Esses documentos só estarão disponíveis para consulta após a conclusão desse método. O `finalizar()`, como pode ser visto no bloco de código a seguir, fecha o objeto `writer`. Uma alternativa é usar o método `commit()` para cada documento. Com isso, a atualização do índice será instantânea. Por outro lado, muitas

chamadas a `commit()` vão degradar significativamente a performance da aplicação porque implica no processamento e otimização do índice e em gravação de arquivos no disco, que são operações custosas.

O método `close()` faz uma chamada ao método `commit()`, ou seja, ambos gravam as inclusões no índice. A diferença é que depois de `close()` aquele `IndexWriter` não poderá mais ser usado, pois estará fechado. Se quiser utilizá-lo novamente terá de chamar o método `open()`. De outra forma, o `commit()` não fecha o índice e permite que o `IndexWriter` continue sendo usado várias vezes.

Por um lado, o `commit()` é muito prático, já que você pode continuar atualizando o índice com o mesmo objeto. Por outro lado, lembre-se de que é uma operação pesada e evite executá-la repetidas vezes. O Lucene tem um recurso de busca em tempo real (NRT) que será discutido no capítulo *5. Principais classes* e resolve essa questão.

```
    public void finalizar() {
        try {
            writer.close();
            diretorio.close();
            //
            logger.info("Total de arquivos indexados: "
                    + totalArquivosIndexados);
            logger.info("Total de bytes indexados (MB): "
                    + totalBytesIndexados / (1024 * 1024));
        } catch (IOException e) {
            logger.error(e);
        }
    }
```

Para realizar os testes, vamos usar a classe `IndexadorArquivosLocaisTest`. Este é um teste unitário e deve ser executado como um *JUnit Test*. Na classe, definimos os diretórios do índice e dos documentos que serão indexados. Note que é apenas uma classe de teste para chamar os métodos do indexador.

Sobre o JUnit precisamos conhecer 3 anotações. Já sabemos que uma classe de teste do JUnit deve ser executada como *JUnit Test* na sua IDE de preferência e não como uma classe Java. Até porque um teste JUnit não tem método *main*. São estas as anotações importantes:

- `@Test` : é o método de teste que será executado.
- `@Before` : método executado antes de cada teste. É usado para inicializar variáveis ou abrir recursos.
- `@After` : método executado após cada teste. É usado para fechar recursos e outras tarefas de finalização.

O *log* gerado durante a indexação se parece com a listagem a seguir. Temos o nome dos arquivos, a quantidade de itens e a quantidade de bytes que foram indexados. Perceba que não é o tamanho do índice e, sim, o tamanho dos arquivos que foram indexados. Como curiosidade, confira o tamanho do índice no seu disco. O tamanho do índice deve ser muito menor que o tamanho dos arquivos.

```
{...}
11:33:52,755  INFO IndexadorArquivosLocais:116 - Arquivo indexado (139
kb): /home/marco/Dropbox/arquivo-1.pdf
11:33:53,398  INFO IndexadorArquivosLocais:116 - Arquivo indexado (398
kb): /home/marco/Dropbox/arquivo-2.pdf
11:33:54,058  INFO IndexadorArquivosLocais:55 - Total de arquivos
indexados: 17
11:33:54,058  INFO IndexadorArquivosLocais:56 - Total de bytes indexados
(MB): 15
```

E agora, finalmente, chegamos à classe que será executada para gerar o índice. Observação: este exemplo usa o diretório `/home/marco/Dropbox/` , que provavelmente não existe na sua máquina, assim, selecione um diretório que contenha os arquivos que pretende indexar. O nome da variável é `DIRETORIO_DOCUMENTOS` .

```
public class IndexadorArquivosLocaisTest {
    private static final Logger logger =
            Logger.getLogger(IndexadorArquivosLocaisTest.class);
```

```
    private static String DIRETORIO_DOCUMENTOS =
            "/home/marco/Dropbox/";
    private static String DIRETORIO_INDICE =
            System.getProperty("user.home")
                    + "/livro-lucene/dropbox";

    @Test
    public void testIndexacao() {
        try {
            IndexadorArquivosLocais indexador =
                    new IndexadorArquivosLocais();
            indexador.setApagarIndice(true);
            indexador.setDiretorioDocumentos(
                    DIRETORIO_DOCUMENTOS);
            indexador.setDiretorioIndice(DIRETORIO_INDICE);
            indexador.setRecursivo(true);
            indexador.inicializar();
            indexador.indexar();
            indexador.finalizar();
        } catch (Exception e) {
            logger.error(e);
        }
    }
}
```

O código mostra como usar a classe de indexação e define alguns parâmetros de controle, por exemplo, apagar o diretório do índice a cada nova execução e indexar com recursão, ou seja, indexar os arquivos do diretório principal e dos seus subdiretórios.

> O documento só ficará disponível para consulta depois de passar pelo método `finalizar()`. Se você indexou poucas dezenas de arquivos, a operação dura poucos segundos. Durante esse tempo, não adianta realizar buscas, porque o índice ainda não existe.

Nos testes executados, com os valores padrão e sem nenhuma otimização, é possível indexar um diretório de 527 arquivos, com

600 MB de dados, em 60 segundos em um computador com 4 processadores Intel i5. O diretório resultante do índice ficou com 20 MB. Claro que a velocidade depende do tamanho dos arquivos e da capacidade de processamento da sua máquina. Arquivos muito grandes e complexos demoram mais para serem analisados.

3.3 Segunda fase: a busca

A busca é a fase mais interessante para o usuário. Enquanto na indexação temos diversas preocupações sobre a origem dos dados, performance e como realizar o melhor pré-processamento, na busca temos apenas que executar uma consulta definida pelo usuário e mostrar os dados recuperados. Exemplo: o usuário quer recuperar todos os documentos que contêm a palavra *java*.

Do lado da aplicação, precisamos traduzir a consulta na linguagem do Lucene, que segue a sintaxe `campo:valor`, `campo:(valor1 OR valor2)` ou ainda `campo:(valor1 AND valor2)`. Essa é uma forma bastante simples de executar consultas e veremos as combinações mais completas no próximo capítulo, que trata deste tema com profundidade.

A classe BuscadorArquivosLocais

Nossas buscas serão feitas através da classe `BuscadorArquivosLocais`. Para facilitar nossa compreensão, ela está separada em quatro blocos de código com funções distintas. Cada um desses blocos será visto logo em seguida neste capítulo, mas, se preferir, o código completo está disponível no GitHub (https://github.com/masreis/exemplos-livro-lucene).

Uma consulta Lucene tem esta estrutura:

1. Acessar o diretório do índice.
2. Executar a consulta do usuário.
3. Processar cada um dos documentos recuperados.
4. Fechar os recursos.

Vamos ao primeiro passo: acessar o diretório do índice. Veja o código:

```
// Abrir o índice e preparar o buscador
Directory diretorio = FSDirectory
        .open(Paths.get(DIRETORIO_INDICE));
IndexReader reader = DirectoryReader
        .open(diretorio);
IndexSearcher searcher = new IndexSearcher(
        reader);
```

No primeiro bloco de código está sendo definido o diretório do índice, além disso, estamos preparando o buscador (`searcher`) para executar as buscas. Precisamos dessas classes:

- `Directory` : classe base que acessa os arquivos do índice no disco do computador.
- `IndexReader` : responsável por acessar a versão atual do índice, ou seja, recupera apenas os documentos indexados até o momento de sua abertura.
- `IndexSearcher` : executa de fato as buscas no índice.

A segunda parte do programa é esta:

```
// Criar e analisar a consulta
QueryParser parser = new QueryParser("",
        new StandardAnalyzer());
Query query = parser.parse(consulta);
logger.info("Consulta analisada-> " + query);
```

Este bloco faz a análise da consulta do usuário, validando sua sintaxe. Para analisar a consulta estamos usando o `StandardAnalyzer` e para validar a sintaxe temos o `QueryParser` . A `Query` armazena a

consulta do usuário, depois de analisada e validada. A última linha mostra o resultado no log. São estas as classes usadas:

- `QueryParser` : analisa a consulta do usuário. O próximo capítulo mostra essa parte em detalhes.
- `StandardAnalyzer` : analisador padrão do Lucene, que faz o pré-processamento básico (retira caracteres especiais e transforma as letras em minúsculas).
- `Query` : representa uma consulta válida que será executada no índice.

Até aqui preparamos o programa para acessar o índice. O bloco a seguir executa a consulta no índice:

```
int QUANTIDADE_DE_ITENS_RETORNADOS = 100;
TopDocs docs = searcher.search(query,
        QUANTIDADE_DE_ITENS_RETORNADOS);
logger.info("Quantidade de itens encontrados: "
        + docs.totalHits);
for (ScoreDoc sd : docs.scoreDocs) {
    Document doc = searcher.doc(sd.doc);
    logger.info(
            "Tamanho: " + doc.get("tamanho"));
    logger.info(
            "Caminho: " + doc.get("caminho"));
    logger.info("Data: " + doc.get("data"));
    logger.info(
            "Extensão: " + doc.get("extensao"));
}
```

Nos exemplos, vamos considerar apenas os primeiros 100 itens encontrados. Poderíamos retornar um número maior de itens, mas considere com precaução utilizar quantidades muito grandes. Será que faz sentido mostrar 1000 documentos em uma página de consulta? Você olha até que página quando utiliza o Google?

Por uma questão de performance, e tudo no Lucene é feito em função da melhor performance, o buscador retorna um *array* apenas com os IDs dos documentos recuperados para a consulta do usuário. Este *array* com os IDs está contido na classe `TopDocs`. A busca não retorna os objetos *Document* que foram indexados, apenas seu ID. Essa é uma preocupação com a performance. Retornar uma coleção de documentos pode representar um problema de memória e processamento para grandes conjuntos de dados. São estas as classes importantes:

- `TopDocs` : contém os IDs dos documentos encontrados pela consulta do usuário.
- `ScoreDoc` : contém o ID específico de um documento.
- `Document` : é o próprio documento que foi indexado.

O resultado representado pelo `TopDocs` está ordenado pela relevância dos itens encontrados, onde o primeiro item é o mais relevante e o último item é menos relevante para aquela consulta. A ordenação do resultado da consulta é baseada no cálculo do TF-IDF e BM25, que foram discutidos no capítulo anterior. O que precisa estar claro é que existe uma ordenação genérica definida por essas medidas. Mais adiante veremos como substituir a ordenação padrão por outras alternativas.

Claro que para o bom aproveitamento do Lucene não é essencial que se saiba em detalhe o que é TF-IDF ou BM25. Contudo, vale a pena dar uma lida sobre essas técnicas. A ideia é muito boa e serve como modelo para uma evolução ou personalização do cálculo para outras situações. Nestes capítulos iniciais ignoramos esses detalhes porque nos capítulos mais avançados mergulharemos em exemplos que tornam essas situações mais claras.

Uma consulta Lucene é eficiente porque utiliza todas essas técnicas e uma parte especialmente interessante é a recuperação de cada documento do índice. Os IDs dos documentos estão armazenados em um *array* numérico, uma estrutura de dados eficiente para esta finalidade, chamada de `ScoreDoc`. Como o nome sugere, esta é uma

classe que contém o identificador do documento e sua pontuação (*score*) no resultado da consulta. É essa pontuação que define sua posição na ordenação do resultado. Os documentos com *score* mais alto são aqueles mais relevantes e ficam no começo do resultado. É a forma padrão de ordenação do Lucene e nos próximos capítulos veremos quais as outras possibilidades.

Agora, sabendo qual a ordem em que os documentos estão organizados, vamos partir para o próximo passo. O resultado da consulta recupera apenas a lista com os identificadores dos documentos encontrados para determinado critério. Em sequência, precisamos carregar o conteúdo textual dos documentos. Essa operação é um pouco mais pesada, por isso não é feito automaticamente pelo Lucene. Fica por sua conta gerenciar e fazer as devidas otimizações. Lembre-se de que operações com Strings em Java são onerosas, tanto do ponto de vista de memória quanto de processamento.

Para recuperar o conteúdo textual de um documento, usamos o método `IndexSearcher.doc(ID)`. O nome do método não é muito sugestivo, mas ele recupera o documento com base em seu ID. A partir desse ponto você pode acessar os valores dos campos. Veja o bloco de código a seguir, que faz exatamente isso. A partir daqui, você pode usar o objeto `doc` com os valores que foram indexados para este documento.

```
for (ScoreDoc sd : docs.scoreDocs) {
    Document doc = searcher.doc(sd.doc);
}
```

O *Document* é a classe que representa cada item ou registro no nosso índice. Funciona como um mapa contendo chave e valor. A chave é o nome do campo, e o valor é o texto que foi indexado. No bloco de código a seguir, os valores de cada campo são impressos no console. Fica claro que o método a ser usado para recuperar os valores é o `Documento.get(nome-do-campo)`. Se o campo não existir,

retorna *null*. Se for um campo multivalorado (o mesmo campo tem vários valores), o `get` retorna o primeiro valor.

```
logger.info(
        "Tamanho: " + doc.get("tamanho"));
logger.info(
        "Caminho: " + doc.get("caminho"));
logger.info("Data: " + doc.get("data"));
logger.info(
        "Extensão: " + doc.get("extensao"));
```

A parte final do programa diz respeito ao fechamento de recursos. Essa é uma parte importante, por isso a ênfase. Nas aplicações com pouco acesso não é tão perceptível, mas a partir do momento em que o projeto está disponível na internet para todos os usuários e o volume de acessos aumenta, é possível (e provável) que em algum momento os recursos do servidor vão acabar. Há um capítulo dedicado aos recursos mais avançados da ferramenta que traz mais detalhes e dicas sobre sistemas em produção.

> Não se esqueça de fechar o IndexReader e o diretório do índice. O Lucene não fecha automaticamente os recursos e isso pode derrubar a aplicação com a exceção *too many open files*, quando há muitos arquivos abertos pelo sistema operacional.

```
// Liberar os recursos
reader.close();
diretorio.close();
```

Analisando consultas com o *Quer Parser*

A primeira fase foi a indexação, onde tornamos os documentos pesquisáveis. Agora podemos executar consultas com base em quaisquer campos indexados do nosso índice. Estão disponíveis no nosso índice os campos `conteudo`, `tamanho`, `data`, `caminho` e `extensao`, então podemos pesquisar qualquer um desses campos para recuperar os documentos correspondentes. O Lucene tem uma

linguagem de consulta com sintaxe simplificada que será explicada a partir de agora. É possível encontrar os documentos que foram alterados em um dia, mês, ano ou intervalo de tempo, documentos com um tamanho específico e, a mais comum, documentos que contêm uma determinada palavra em seu conteúdo. Basicamente, é a mesma coisa que você faz com o Google e Bing.

Para criar uma consulta, o mais rápido é usar a sintaxe `campo:valor`. Por exemplo: `conteudo:algumaPalavraNoTexto`. Essa sintaxe vai retornar todos os documentos que contenham `algumaPalavraNoTexto` no campo `conteudo`. Essa é a sintaxe clássica utilizada pelo analisador de consultas do Lucene e que permite realizar a maioria das buscas de uma forma simples.

Nosso primeiro exemplo de consulta pode usar o campo `conteudo`, com o conteúdo textual extraído dos seus arquivos. Nossa consulta é `conteudo:java`, onde buscamos pelos documentos que contêm a palavra `java` em seu no campo `conteudo`. Esse teste é feito na classe `BuscadorArquivosLocaisTest`.

```
@Test
public void testConsultaPorConteudo() {
    BuscadorArquivosLocais buscador =
            new BuscadorArquivosLocais();
    String consulta = "conteudo:java";
    buscador.buscar(consulta);
}
```

Evidentemente, nem todas as consultas são tão simples. Para refinar o resultado, geralmente a consulta tem mais de uma palavra. Agora vamos buscar por `ciência da informação`. Quando não for explicitamente indicado, o padrão é utilizar o operador lógico *OU*. Com isso, a busca retorna os documentos que contêm qualquer um dos 3 termos, ou seja, os documentos que contêm, em qualquer posição, as palavras `ciência` *OU* `da` *OU* `informação`. É importante notar que para buscar mais de uma palavra é preciso usar os parênteses.

Você percebe que a figura central para processar as consultas do usuário é o `QueryParser`. Sua tarefa é analisar a expressão de consulta informada pelo usuário e transformá-la em um objeto da classe `Query`. Seu método mais importante é o `parse`, que retornará um objeto `Query` que representa a expressão de consulta desejada. Para consultas simples, naturalmente o retorno é apenas um objeto `Query` simples, mas o `QueryParser` é capaz de analisar expressões complexas, desde que siga sua sintaxe. Veremos esses detalhes no capítulo sobre buscas.

> A contração *da* (preposição + artigo) é um termo de baixa relevância, ou *stop word*. Com isso, as consultas para `ciência da informação` e `ciência informação` trazem praticamente o mesmo resultado.

```
@Test
public void testConsultaPorConteudo() {
    BuscadorArquivosLocais buscador =
            new BuscadorArquivosLocais();
    String consulta = "conteudo:(java AND jsf)";
    buscador.buscar(consulta);
}
```

Nossa consulta `conteudo:(ciência da informação)`, depois de analisada pelo `StandardAnalyzer`, será traduzida para `conteudo:ciência conteudo:da conteudo:informação`, que é outra forma de escrever a mesma consulta. Por fim, o parser encapsulará tudo isso em um objeto do tipo `Query` que será passado para o `IndexSearcher` executar a busca no índice.

> Para conferir como sua consulta foi analisada, use a instrução `logger.info("Consulta analisada -> " + query)`.

Agora que temos uma consulta com mais de uma palavra, faz sentido a busca exata utilizando aspas. Como o nome sugere,

somente retorna os documentos que contêm exatamente o termo entre aspas. É muito mais restritiva que a anterior e deve trazer uma quantidade igual ou menor de documentos: `consulta = "conteudo:(\"ciência da informação\")"`.

Na busca exata com aspas, os parênteses são desnecessários porque consideramos todas as palavras como sendo apenas um termo. A próxima consulta, mesmo sem parênteses, é exatamente igual à anterior: `consulta = "conteudo:\"ciência da informação\""`.

Experimente consultar por outros campos e valores. Por exemplo com o campo `extensao` para verificar quais arquivos têm extensão `doc`. Assim:

```
@Test
public void testConsultaPorExtensao() {
    BuscadorArquivosLocais buscador =
            new BuscadorArquivosLocais();
    String consulta = "extensao:pdf";
    buscador.buscar(consulta);
}
```

A sintaxe aceita os operadores lógicos `AND`, `OR` e `NOT` para conectar critérios de busca. São os operadores que você, como programador, está acostumado a usar. O operador `AND` indica que os dois termos são obrigatórios. O operador `OR` implica o resultado pode conter qualquer um dos termos, dessa forma, o resultado traz documentos que contêm um ou outro termo indicado. O último operador, o `NOT`, indica a negação, quer dizer, aquela palavra não pode existir no resultado da consulta.

Veja o primeiro exemplo, que traz os documentos que contêm obrigatoriamente os dois termos (java AND jsf). Esta consulta garante que todos os itens do resultado da consulta contêm as palavras `java` e `jsf`: `consulta = "conteudo:(java AND jsf)"`.

Neste caso é obrigatório o uso de parênteses e o operador (AND, NOT, OR) deve ser escrito com letras maiúsculas. A consulta `conteudo:java AND jsf` é diferente de `conteudo:(java AND jsf)`.

O operador `OR` indica que o resultado pode conter qualquer um dos termos, isto é, cada item do resultado pode conter a palavra `java` ou a palavra `jsf`. Os documentos que contêm os dois termos (`jsf` e `java`) têm relevância maior e estarão nas primeiras posições do resultado: `consulta = "conteudo:(java OR jsf)"`.

Por fim, o operador `NOT` representa a negação de um termo. Desta forma, o resultado da consulta não trará nenhum item que contém aquela palavra. O exemplo a seguir recupera todos os documentos que, em seu conteúdo, têm a palavra `java`, porém, não contêm de forma alguma a palavra `jsf`: `consulta = "conteudo:(java NOT jsf)"`.

O operador `NOT` não pode ser usado isoladamente e deve sempre estar associado a outro termo. Exemplo: a consulta `conteudo:(NOT jsf)` não traz nenhum resultado porque o termo com `NOT` está sozinho.

Consultando intervalos de valores

Para campos como `data`, uma boa opção é usar a busca por intervalo. Essa busca é utilizada para intervalos de valores alfanuméricos. A consulta por intervalos numéricos será vista no próximo capítulo.

Nós indexamos os documentos usando o formato *yyyyMMdd* para data, que é o formato usado nos EUA. Essa data em formato invertido é essencial para garantir a correção no resultado das nossas consultas. Agora, para encontrar os arquivos modificados

em 2016, a consulta seria assim: `consulta = "data:[20160101 TO 20161231]"`.

Se não quiser usar a precisão da data com *yyyyMMdd*, pode filtrar apenas por *yyyyMM* ou até mesmo por *yyyy*. Para encontrar os arquivos modificados esse ano utilizando o formato *yyyy*, siga esse modelo: `consulta = "data:[2016 TO 2017]"`.

Devemos atentar para os detalhes da sintaxe. Perceba que o `TO` é maiúsculo e que os colchetes (`[]`) indicam que as datas são incluídas na consulta. Para excluir as datas utilize chaves (`{}`). A consulta a seguir encontra, novamente, os arquivos modificados durante esse ano, mas exclui do resultado os arquivos modificados no dia `20160101` e `20161231`, desta forma: `consulta = "data:{20160101 TO 20161231}"`.

Vale lembrar de que esse é a sintaxe para busca de intervalos alfanuméricos, por isso usamos a data em formato invertido (`yyyyMMdd` em vez de `dd/MM/yyyy`). Com esse formato, podemos pesquisar intervalos de datas e alfanuméricos (palavras). Se quiser encontrar os nomes entre `ana` e `beatriz`, faça desta forma: `consulta = "conteudo:[ana TO beatriz]"`.

3.4 Removendo documentos do índice

O ciclo de vida de uma informação é finito. Quando um registro não é mais necessário ele deve ser removido da base, inclusive para economizar espaço. Da mesma forma acontece com o Lucene. Quando um documento não é mais necessário, devemos removê-lo do índice. É o que veremos nesta seção.

Uma exclusão no Lucene implica na alteração do índice, de onde aquele documento será removido definitivamente. O método chamado para excluir documentos no Lucene é o `IndexWriter.deleteDocuments`, que pode receber como parâmetro uma

lista de objetos `Query` ou `Term`. Portanto, para excluir um documento do índice você envia como parâmetro uma consulta.

Exemplo: considere a exclusão do documento com o nome `c:\algum_diretorio\teste.txt`. Para excluí-lo do índice precisamos usar a consulta `caminho:"c:\algum_diretorio\teste.txt"`. No nosso sistema temos que cada arquivo tem um único caminho. Com essa consulta, então, garantimos que será removido apenas um único arquivo. Contudo, podemos expandir a remoção a todos os arquivos de texto assim: `extensao:txt`. Aqui começam os problemas.

Nem preciso lembrar que uma remoção descuidada pode ser traumática. Se por um infortúnio qualquer você enviar uma consulta mais abrangente como `conteudo:a*`, você vai excluir todos os documentos que contenham a letra `a`, ou seja, você praticamente vai matar todo o seu índice.

Assim como na indexação, a exclusão definitiva acontece apenas a partir do momento em que for chamado o `IndexWriter.commit()` ou `IndexWriter.close()`. Vamos rever esse ponto: o `commit()` grava as alterações no índice e permite que o `IndexWriter` continue sendo usado. O `close()` faz uma chamada ao `commit()` e fecha o `IndexWriter`, impedindo seu uso até que você use o método `open`. Depois do `commit()` ou `close()` as alterações são efetivadas no índice e aqueles documentos excluídos não serão mais retornados pelo `IndexSearcher`.

Atenção: a exclusão é uma tarefa crítica, com muitos pontos de falha e problemas com a preservação dos dados. Tanto que alguns sistemas não têm sequer exclusão dos registros, preferindo atualizar uma *flag* indicando que aquela linha está desativada ou oculta. É um fator para melhorar a segurança e permitir auditoria de aplicações críticas, evitando fraudes. É uma estratégia válida e inteligente que pode ser usada no Lucene.

A classe que faz exclusões no nosso índice é a `ExcluirDocumentoIndiceTest`, disponível no GitHub

(https://github.com/masreis/exemplos-livro-lucene/). Veja a listagem dessa classe a seguir. Nós temos um método para inicializar o `IndexWriter`, um método de teste (`test01ExclusaoArquivo`), o método `finalizar` e, por fim, o `verificarQuantidadeDocumentos`. Vamos detalhar cada um dos blocos na sequência.

```
@FixMethodOrder(MethodSorters.NAME_ASCENDING)
public class ExcluirDocumentoIndiceTest {
    private static String DIRETORIO_INDICE = System
            .getProperty("user.home")
            + "/livro-lucene/indice";
    private static final Logger logger = Logger
            .getLogger(ExcluirDocumentoIndiceTest.class);
    private Analyzer analyzer;
    private Directory diretorio;
    private IndexWriterConfig conf;
    private IndexWriter writer;

    @Before
    public void inicializarWriter() throws IOException {
        analyzer = new StandardAnalyzer();
        diretorio = FSDirectory
                .open(Paths.get((DIRETORIO_INDICE)));
        conf = new IndexWriterConfig(analyzer);
        writer = new IndexWriter(diretorio, conf);
    }

    @Test
    public void test01ExclusaoArquivo()
            throws IOException {
        // Termo que define o documento que será excluído
        // Observação:
        // este caminho deve ser de um arquivo existente
        Term termoParaExclusao = new Term("caminho",
                "/home/papers/proposta-reforma.pdf");
        verificarQuantidadeDocumentos(termoParaExclusao);
        // Verifica a quantidade de documentos antes da exclusão
        writer.deleteDocuments(termoParaExclusao);
        writer.commit();
        // Verifica a quantidade de documentos depois da exclusão
```

```
        verificarQuantidadeDocumentos(termoParaExclusao);
    }

    @After
    public void finalizar() throws IOException {
        writer.close();
        diretorio.close();
    }

    private void verificarQuantidadeDocumentos(
            Term termoParaExclusao) throws IOException {
        IndexReader reader = DirectoryReader
                .open(diretorio);
        IndexSearcher searcher = new IndexSearcher(reader);
        TopDocs docs = searcher.search(
                new TermQuery(termoParaExclusao), 1);
        logger.info("Quantidade de documentos encontrados: "
                + docs.totalHits);
        // Verifica se a consulta retorna apenas um documento
        if (docs.totalHits > 1) {
            // Aconteceu algum problema
            logger.warn(
                    "Essa exclusão é potencialmente perigosa");
        }
        //
        logger.info("NumDocs: " + reader.numDocs());
        logger.info("MaxDoc: " + reader.maxDoc());
        logger.info(
                "HasDeletions: " + reader.hasDeletions());
        reader.close();
    }
}
```

Os requisitos para exclusão de documentos do índice são:

- Critério para exclusão. Pode ser um único documento, apenas os documentos com extensão PDF ou documentos com data do ano passado. Isso será definido no critério de exclusão.
- `IndexWriter` : é o responsável por gravar (e excluir) documentos no índice.

Nosso objetivo é excluir o documento que contém o caminho indicado (`/home/marco/Dropbox/tese.pdf`) e para tanto vamos usar uma `TermQuery` para expressar esse critério. Assim como eu sei que existe aquele documento específico, o `IndexWriter` também sabe e vai encontrá-lo no índice para fazer sua exclusão com o método `deleteDocuments` . Caso não encontre nenhum documento que combine com os parâmetros nada acontece, sem problemas. Vale recordar que essa alteração, a exclusão, só estará visível para os usuários do sistema depois do `commit` ou `close` . Na sequência podemos conferir como ficou. Note que temos duas chamadas para o método `verificarQuantidadeDocumentos(termoParaExclusao)` , verificando o estado do índice antes e depois da exclusão.

Mas e se você especificar erroneamente o critério de exclusão? Por exemplo, se o arquivo para exclusão não existir, o que o sistema deveria fazer? E se você especificar um critério que exclua mais documentos do que o esperado? Para esses casos foi criado o método `verificarQuantidadeDocumentos` , onde fazemos uma validação mínima antes da exclusão. Este é o momento em que você pode fazer algum tratamento. Depois do `commit` os documentos desaparecem e não há recuperação. Note que verificamos a quantidade de documentos no índice duas vezes, antes e depois da exclusão.

O método `verificarQuantidadeDocumentos` confere se o critério encontra um e somente um documento. Claro, neste contexto específico que definimos. No nosso exemplo vamos excluir apenas um documento de cada vez. No final, o método imprime 3 informações importantes:

- A quantidade de documentos que existe no índice.
- O identificador do maior documento indexado.
- Se o índice sofreu exclusões.

Perceba que se encontrar mais de um item é porque nosso critério não está correto. Neste caso o método imprime uma mensagem de alerta no console. É o que indica a instrução `if (docs.totalHits > 1)` . Em um sistema real, este é o ponto onde colocaríamos uma

proteção contra exclusões acidentais. Mas é claro que em outra situação pode ser que você queira excluir muitos documentos simultaneamente. Fica a dica de tomar os devidos cuidados para não destruir o índice.

Nas primeiras linhas do método temos um `IndexReader`, assim como no exemplo do buscador. Este `IndexReader` reconhece o estado *atual* do índice no momento de sua abertura. É uma informação importante porque este objeto específico não reconhece atualizações que ocorrem depois de ter sido aberto. Se outras *threads* separadas atualizarem o índice, este objeto não perceberá essas diferenças. O que vale para este `IndexReader` é a situação do índice no momento de sua abertura e, para ver as atualizações, você precisa fechar e abrir novamente o recurso. Não é um grande problema aqui no nosso pequeno exemplo, contudo precisa ser considerado em sistemas com múltiplos usuários, onde o `IndexReader` fica aberto por muito tempo ou é reutilizado por várias partes da aplicação.

Depois de executar o teste teremos o resultado que aparece a seguir. Meu índice contém 527 documentos, como pode ser conferido na primeira listagem. Esta é a primeira verificação da quantidade de documentos do índice *antes da exclusão*. Aqui diz que o critério de exclusão encontrou um documento e que o índice tem 527 documentos, sendo que o número do último documento é exatamente o de número 527, além de que não há ainda exclusões no índice. Veja:

```
17:43:34,745  INFO ExcluirDocumentoIndiceTest:69 - Quantidade de
documentos encontrados: 1
17:43:34,754  INFO ExcluirDocumentoIndiceTest:76 - NumDocs: 527
17:43:34,754  INFO ExcluirDocumentoIndiceTest:75 - MaxDoc: 527
17:43:34,754  INFO ExcluirDocumentoIndiceTest:77 - HasDeletions: false
```

A listagem de *log* a seguir mostra o resultado *depois da exclusão* do documento. Note que a consulta já não encontrou o documento, pois ele foi excluído com a instrução `writer.deleteDocuments(termoParaExclusao);`. Nosso método funcionou a

contento, o que pode ser confirmado com o número de documentos do índice que agora é 526, quer dizer, tem 1 item a menos. Contudo, o número do último documento ainda é o 527 e isso não se modifica porque de fato indexamos 527 documentos. Para finalizar, confira que nesta instância do `IndexWriter` a opção `HasDeletions` está `true`, ou seja, ele sabe que o índice sofreu exclusões.

```
17:43:34,871  INFO ExcluirDocumentoIndiceTest:69 - Quantidade de
documentos encontrados: 0
17:43:34,871  INFO ExcluirDocumentoIndiceTest:76 - NumDocs: 526
17:43:34,871  INFO ExcluirDocumentoIndiceTest:75 - MaxDoc: 527
17:43:34,871  INFO ExcluirDocumentoIndiceTest:77 - HasDeletions: true
```

Se rodar novamente o mesmo teste o resultado não se altera porque o documento já foi excluído. Teste com outros itens que foram indexados e verifique se o número de documentos do índice diminui. O esperado é que isso aconteça, se o caminho indicado no critério tiver sido indexado. A listagem a seguir mostra o que deveria acontecer se eu executasse o teste com outro documento que existe no índice. Veja que o número de documentos do índice agora é 525, mas o número do último documento continua sendo 527. Este é o comportamento esperado da aplicação.

```
17:43:34,871  INFO ExcluirDocumentoIndiceTest:69 - Quantidade de
documentos encontrados: 1
17:43:34,871  INFO ExcluirDocumentoIndiceTest:76 - NumDocs: 525
17:43:34,871  INFO ExcluirDocumentoIndiceTest:75 - MaxDoc: 527
17:43:34,871  INFO ExcluirDocumentoIndiceTest:77 - HasDeletions: true
```

3.5 Atualizando documento no índice

O conteúdo de um texto costuma mudar. Isso é natural e até mesmo esperado em algumas situações. Mesmo notícias no site de um jornal costumam ser atualizadas no decorrer do dia, com correções ou adições. Mas o índice reflete apenas o conteúdo no momento da indexação. Se um documento for atualizado depois da indexação, o

índice estará obsoleto. Se o documento sofreu alterações ele deve ser atualizado também no índice. E não existe uma forma fácil de dizer isso: o Lucene não atualiza documentos.

Até existe o método `IndexWriter.updateDocument`, todavia, na prática o que ele faz é excluir e adicionar o documento novamente no índice. Se no exemplo anterior nós excluímos um documento, agora vamos fazer uma inclusão, isto é, teremos uma atualização. O código está na classe `IndexarDocumentoIndiceTest`. A listagem completa desta classe está na sequência. É uma combinação de todas as outras classes que vimos até agora. Os métodos da classe são:

- `testIndexarArquivo`: o principal método da nossa classe e que efetivamente inclui o documento. Vamos colocar de volta no índice o arquivo `/home/marco/Dropbox/tese.pdf`.
- `verificarQuantidadeDocumentos`: é o mesmo do exemplo anterior. Está aqui para vermos o comportamento do índice durante a inclusão.
- `inicializar`: abre o diretório do índice.
- `fechar`: fecha o diretório do índice.

```
public class IndexarDocumentoIndiceTest {
    private static final Logger logger = Logger
            .getLogger(IndexarDocumentoIndiceTest.class);
    private static String DIRETORIO_INDICE = System
            .getProperty("user.home")
            + "/livro-lucene/indice";
    private Directory diretorio;

    @Test
    public void testIndexarArquivo() {
        try {
            String nomeArquivo = "/home/marco/proposta-reforma.pdf";
            Term termoParaExclusao = new Term("caminho",
                    nomeArquivo);
            //
            verificarQuantidadeDocumentos(
                    termoParaExclusao);
            IndexadorArquivosLocais indexador =
```

```
                new IndexadorArquivosLocais();
        indexador.setDiretorioIndice(DIRETORIO_INDICE);
        indexador.inicializar();
        indexador.indexarArquivo(new File(nomeArquivo));
        indexador.finalizar();
        //
        verificarQuantidadeDocumentos(
                termoParaExclusao);
    } catch (Exception e) {
        logger.error(e);
    }
}

private void verificarQuantidadeDocumentos(
        Term termoParaExclusao) throws IOException {
    IndexReader reader = DirectoryReader
            .open(diretorio);
    IndexSearcher searcher = new IndexSearcher(reader);
    TopDocs docs = searcher.search(
            new TermQuery(termoParaExclusao), 1);
    logger.info("Quantidade de documentos encontrados: "
            + docs.totalHits);
    // Verifica se a consulta retorna apenas um documento
    if (docs.totalHits > 1) {
        // Aconteceu algum problema
        logger.warn(
                "Essa exclusão é potencialmente perigosa");
    }
    //
    logger.info("NumDocs: " + reader.numDocs());
    logger.info("MaxDoc: " + reader.maxDoc());
    logger.info(
            "HasDeletions: " + reader.hasDeletions());
    reader.close();
}

@Before
public void inicializar() throws IOException {
    diretorio = FSDirectory
            .open(Paths.get((DIRETORIO_INDICE)));
}
```

```
    @After
    public void fechar() throws IOException {
        diretorio.close();
    }
}
```

O método que importa aqui é o `testIndexacao`. Os demais são, novamente, apenas acessórios. Vamos aos detalhes. O `testIndexacao` começa com uma chamada ao método `verificarQuantidadeDocumentos`. Isso é para garantir que aquele documento não está no índice. O `verificarQuantidadeDocumentos` imprime o seguinte *log* no console:

```
21:22:39,880  INFO IndexarDocumentoIndiceTest:50 - Quantidade de
documentos encontrados: 0
21:22:39,884  INFO IndexarDocumentoIndiceTest:57 - NumDocs: 525
21:22:39,884  INFO IndexarDocumentoIndiceTest:58 - MaxDoc: 527
21:22:39,884  INFO IndexarDocumentoIndiceTest:59 - HasDeletions: true
```

Significa que não encontrou aquele documento no índice (`/home/marco/Dropbox/tese.pdf`). Depois, imprime o número de documentos indexados, número do último documento indexado, que continua sendo 527 e que o índice já sofreu exclusão.

Continuando nossa explicação, vamos criar um objeto do tipo `IndexadorArquivosLocais`, aquele mesmo que foi usado anteriormente, mas, dessa vez, só precisamos chamar o método `indexador.indexarArquivo(new File(nomeArquivo))`. Esta chamada faz com que o arquivo indicado seja incluído no índice, reutilizando o código da classe. Na sequência temos o método `indexador.finalizar()` e outra chamada ao `verificarQuantidadeDocumentos`.

E agora, como será que está nosso índice? Teoricamente, mais um item foi adicionado. É o que esperamos. Para saber se é verdade, analise o *log* impresso no console. Tem que ser algo similar a esse:

```
21:22:41,287  INFO IndexadorArquivosLocais:116 - Arquivo indexado (139
kb): /home/marco/Dropbox/tese.pdf
21:22:41,650  INFO IndexadorArquivosLocais:55 - Total de arquivos
indexados: 1
21:22:41,650  INFO IndexadorArquivosLocais:56 - Total de bytes indexados
(MB): 0
21:22:41,668  INFO IndexarDocumentoIndiceTest:50 - Quantidade de
documentos encontrados: 1
21:22:41,669  INFO IndexarDocumentoIndiceTest:57 - NumDocs: 526
21:22:41,669  INFO IndexarDocumentoIndiceTest:58 - MaxDoc: 528
21:22:41,670  INFO IndexarDocumentoIndiceTest:59 - HasDeletions: true
```

Após a execução do novo teste de indexação, o índice conta com 526, ou seja, foi adicionado 1 novo documento. Outra mudança foi o número máximo de documentos, que agora é 528, ou seja, até agora foram indexados 528 itens, mesmo que um deles tenha sido indexado 2 vezes. Para fechar, o atributo `HasDeletions` continua com valor `true` , indicando, mais uma vez, que o índice sofreu alterações.

Resumo

Neste capítulo criamos um Sistema de Recuperação de Informações (SRI) completo tendo como base os arquivos locais do seu computador. É totalmente funcional e extensível a situações mais complexas. Vimos como foi feita a implementação do processo de indexação e de busca com a criação das classes `IndexadorArquivosLocais` e `BuscadorArquivosLocais` , que são reutilizáveis em outros projetos.

Percebemos que alguns pontos são padronizados, como abrir um `IndexWriter` e um `IndexReader` , bem como o fechamento dos recursos após seu uso. O procedimento vai ser sempre igual. E as recomendações também. A principal recomendação é que você evite usar o método `commit` muitas vezes. A forma mais indicada é fazer um `commit` em intervalos de tempo regulares, a cada minuto ou a cada 10 minutos, por exemplo.

Ainda neste capítulo fizemos consultas básicas e algumas mais elaboradas, utilizando os principais operadores lógicos (`AND` , `OR` e `NOT`), que são suficientes para situações simples com poucos campos.

Chegamos ao final do capítulo com as operações de exclusão e inclusão de documentos individuais no índice. Essa é a parte de manutenção do índice, onde itens antigos são retirados ou atualizados. Neste ponto verificamos o comportamento do índice durante as operações de exclusão e inclusão, com a mudança nos valores do `NumDocs` , `MaxDocs` e `HasDeletions` , atributos que indicam o estado atual do índice, ainda que de forma simplificada.

Com isso finalizamos nosso primeiro Sistema de Recuperação de Informações (SRI). Todas as operações foram implementadas. Continuaremos no próximo capítulo com o conteúdo avançado para construção de consultas complexas. Vamos detalhar como funciona o mecanismo de consulta do Lucene, conhecer as variações da sintaxe e apresentaremos a API de consultas, uma alternativa para construir buscas dinamicamente.

Capítulo 4
Tipos de busca

A busca é uma funcionalidade essencial em qualquer sistema e neste capítulo vamos nos aprofundar nas possibilidades oferecidas pelo Lucene. O objetivo é consultar dados de várias formas diferentes, atendendo a situações simples e complexas. Com isso, exploramos a grande maioria das opções de consultas disponíveis.

Existem diversos tipos de busca. A mais simples é a busca por uma palavra-chave, quando você pesquisa por algum assunto de seu interesse no buscador, por exemplo, pelo termo *economia* no Google. Outra opção comum é a combinação de palavras, como em *curso de java*, *direito civil* ou *receita de pão caseiro*.

Mas há também alternativas mais complexas, como a busca por sinônimos, onde você pesquisa por *tarefa* e o buscador considera *atividade*, *função* e *exercícios*, que são palavras semelhantes. Ainda temos as buscas baseadas em padrões, como CEP, CPF, número de telefone, e-mail etc. Veremos em detalhe cada uma dessas possibilidades, e várias outras.

O código de um buscador com Lucene pode ser escrito de 2 formas diferentes: com a sintaxe clássica e com a API. Elas têm o mesmo funcionamento (ou quase, como veremos depois) e a performance é igual. O que muda é a forma como escrevemos o código-fonte. Para ficar claro veremos a implementação das consultas usando as 2 formas, inicialmente temos uma lista de consultas com a sintaxe clássica e na sequência as mesmas consultas usando a API.

4.1 Comparação com uma consulta SQL

Aqui vale uma comparação com as consultas SQL. Uma boa parte do que o Lucene faz é possível com os bancos de dados mais modernos. A maioria deles implementa sua solução para busca textual, porém, de forma proprietária, ou seja, só funciona dentro daquele produto. O PostgreSQL, por exemplo, tem um sistema de busca textual, mas só funciona com os dados que estão no próprio banco. Sem contar que não é possível adicionar novas funcionalidades ao seu buscador.

Você estará limitado aos recursos disponíveis naquele banco de dados, que estão aquém do que se consegue fazer uma ferramenta especializada em busca textual, como o Lucene. Ademais, para cobrir as várias necessidades de consulta de uma aplicação, devemos escrever consultas SQL complicadas.

Entenda que uma boa parte do tempo de desenvolvimento de um software é utilizado escrevendo comandos SQL. Mesmo com os frameworks para mapeamento objeto-relacional (JPA, Hibernate, Spring JDBC, MyBatis) que facilitam o trabalho, ainda precisamos escrever consultas para encontrar os dados de que o usuário precisa.

Os requisitos para módulos de consulta de um software tendem ao infinito. E é fácil entender o motivo. A ideia de se adquirir um sistema é exatamente essa: agilizar ou facilitar o trabalho do usuário. E este usuário está cada vez mais exigente no que diz respeito às buscas, em grande parte porque está acostumando com os buscadores da internet. Quem usa o Google uma vez não quer saber de sistemas com buscas limitadas. As buscas simplesmente por palavra-chave estão fadadas a desaparecer. A boa notícia é que o mecanismo de busca do Lucene atende com eficiência a essa necessidade.

Novamente, é preciso entender que o Lucene é uma ferramenta complementar e não substitui o seu banco de dados relacional. O objetivo do Lucene não é nem nunca será substituir o Oracle ou SQL Server, até porque o modelo tradicional de consulta fornecido

pelo banco através da SQL é eficiente em muitas determinadas situações. Por exemplo:

- Encontrar uma pessoa pelo CPF
- Encontrar um livro pelo seu ISBN
- Encontrar um usuário pelo e-mail
- Encontrar as cidades de um estado
- Encontrar as operações bancárias realizadas semana passada
- Encontrar quais produtos fazem parte de uma determinada categoria
- Encontrar as vendas com valor entre $1.000 e $2.000

Existe um padrão neste tipo de pesquisa, consegue entender qual é? Em todos esses casos, o usuário está usando critérios simples, baseados em um valor determinado e conhecido, como uma coluna que armazena o número do CPF ou outra coluna que armazena apenas e-mails. Acontece que situações mais complexas ou situações que envolvem grandes volumes de dados podem não ter um bom resultado quando se utiliza banco relacional. E é exatamente onde temos o cenário ideal para utilizar o Apache Lucene.

Vamos considerar outras situações de um cenário de busca:

- Buscar as notícias que tratam do assunto *olimpíadas*, mas não das *olimpíadas de inverno*.
- Buscar em todos os campos ou em todos os campos de todas as tabelas.
- Os registros ou a consulta do usuário apresentam de erro de grafia. Português é um idioma complicado e existem termos como *traz* e *trás*, *porque* e *por que*.
- Buscar as variações de nomes como em *Giovane* e *Geovane*, *Marcus* e *Marcos*, *Rafael* e *Raphael*. Imagine quantas grafias diferentes de *Wellington* existem no mundo.
- Em um sistema de e-commerce, o usuário procura por TV e quer encontrar seus sinônimos como televisão, televisor e suas variações como TV de LED, TV de LCD, TV de 32 polegadas.

- Buscar questões em fórum de discussão, portal de internet ou intranet, review de produtos e sites de viagem.
- Buscar linhas específicas em arquivos com bilhões de linhas de *log*.

À medida que a quantidade de registros de um banco de dados aumenta, há degradação na performance das buscas. Pesquisas de intervalos de valores (aquelas com o *between* do SQL), pesquisas com o agrupamento de valores (o famoso *group by* do SQL) ou em texto livre (aquelas com *like* do SQL) possivelmente não responderão com a velocidade esperada. Por isso criamos sistemas especializados em busca. Com um motor de busca como o Lucene, o sistema terá a habilidade de encontrar fácil e rapidamente os itens que o usuário procura.

4.2 Sintaxe clássica de buscas

Nas primeiras versões do Lucene, as consultas eram executadas com um conjunto de palavras reservadas que chamamos de sintaxe clássica. Com ela é possível definir critérios de busca para a recuperação de documentos. Apenas para relembrar, esses critérios seguem a estrutura `nome-do-campo:valor` como no exemplo `nome:jose`. Você quer recuperar todos os documentos que contêm a palavra `jose` no campo `nome`.

Antes de executar a consulta, porém, o Lucene precisa realizar algumas validações. Isso é feito com o `QueryParser`, uma classe importante desta biblioteca. A função do `QueryParser` é analisar a consulta escrita pelo programador, validar a sintaxe e criar um objeto do tipo `Query` com os comandos.

Para fazer a análise sintática, o `QueryParser` usa a biblioteca JavaCC (*Java Compiler Compiler*) que valida os comandos. Caso a expressão de consulta inclua algum erro de sintaxe, será lançada a

exceção `ParseException`. Então, atenção para parênteses balanceados e palavras reservadas do Lucene.

Os exemplos a seguir estão na classe `BuscadorSintaxeClassicaTest`. A listagem do código parcial está logo abaixo e será usada para a primeira parte dos exemplos deste capítulo. Para todos os casos, lembre-se de que os campos que indexamos são `conteudo`, `tamanho`, `data` e `caminho`. São essas as opções que temos neste momento para pesquisar. Veja o código a seguir e use-o para os demais exemplos do capítulo.

```
public class BuscadorSintaxeClassicaTest {
    private static final Logger logger = Logger
            .getLogger(BuscadorSintaxeClassicaTest.class);
    public void testConsultaSintaxeClassica() {
        logger.info("Sintaxe clássica");
        String consulta = "conteudo:java";
        BuscadorArquivosLocais buscador =
                new BuscadorArquivosLocais();
        buscador.buscar(consulta);
    }
    //{...}
}
```

Aqui vemos como usar a sintaxe clássica do Lucene com a consulta `conteudo:java`, isto é, todos os documentos que contêm a palavra `java` em seu conteúdo. Continuamos usando a classe `BuscadorArquivosLocais` do capítulo anterior, dessa vez com critérios mais complexos de busca. O ponto central é mostrar a sintaxe clássica e a análise feita pelo `QueryParser`. Após a análise da expressão de consulta digitada desejada, o `QueryParser` retorna um objeto do tipo `Query`. É isso que precisamos para recuperar os documentos do índice. Esse passo é realizado pelo código:

```
    QueryParser parser = new QueryParser("",
            new StandardAnalyzer());
    Query query = parser.parse(consulta);
```

Ainda analisando a `BuscadorArquivosLocais`, a sintaxe para inicializar nosso *parser* tem dois parâmetros construtores. O primeiro é o campo que desejamos pesquisar. Perceba que nos nossos exemplos estamos usando uma String vazia para o campo pesquisado: `...new QueryParser("", analisador);`. É uma construção válida, desde que você informe o campo na expressão de consulta: `parse("conteudo:(ciência da informação)");`.

Portanto, temos duas formas de criar uma `Query` com o `QueryParser`. Compare as alternativas:

```
QueryParser parser = new QueryParser("",
    analisador);
Query query = parser.parse(
        "conteudo:(ciência da informação)");
```

Este bloco poderia ser reescrito de outra forma com o mesmo resultado:

```
QueryParser parser = new QueryParser("conteudo",
    analisador);
Query query = parser.parse("ciência da informação");
```

Prefiro a primeira opção porque você pode reutilizar o *parser* para outras consultas depois. No segundo exemplo, o *parser* já está apontando para um campo específico e apenas o parâmetro de consulta pode ser alterado.

4.3 Buscas com a sintaxe clássica

Nesta seção mostraremos vários tipos de busca, desde as mais simples até as muito elaboradas. O primeiro passo para a criação do nosso sistema de buscas personalizado é conhecer as funcionalidades do Lucene, o que será feito a partir de exemplos. Por uma questão de praticidade, vou omitir o código Java e será mostrada apenas a consulta do Lucene. Na classe Java, você deve

substituir o valor da variável `consulta` que está na classe/método `BuscadorArquivosLocais.testConsultaSintaxeClassica`.

conteudo:java

Este é o tipo mais natural de busca e recupera os documentos que contêm a palavra `java` no campo `conteudo`. O campo `conteudo` inclui o texto do arquivo e é um bom ponto de partida para nosso trabalho de pesquisa. Da mesma forma, para buscar outras palavras use a sintaxe `campo:palavra`, como em `conteudo:economia`, `conteudo:política` ou `conteudo:futebol`. O importante é que essa sintaxe comporta a pesquisa por uma única palavra.

conteudo:java AND data:"20180606"

Recupera os documentos que contêm a palavra `java` em seu conteúdo e que tenham sido indexados na data `06/06/2018`. É importante lembrar que a data deve seguir o padrão americano, que é *yyyyMMddHHmmss*. Além disso, você pode consultar com uma precisão menor, como *yyyyMMdd*, que foi a usada na indexação dos nossos documentos. Então, agora ficou claro porque temos de usar a data no formato especificado.

conteudo:(java -cdi)

Esta é a busca disjuntiva, que recupera os documentos que contêm uma palavra e que não contêm a outra, formando a disjunção, ou seja, o contrário de junção. No exemplo procuramos documentos com `java` e que *não* contêm a palavra `cdi` em seu conteúdo. Você pode usar o recurso com mais termos, como em `conteudo:(java -rest -jpa)`. Neste caso estamos procurando por documentos que contenham a palavra `java` e que não contenham os termos `rest` e `jpa`.

A sintaxe clássica permite reescrever a consulta desta forma: `conteudo:java -conteudo:cdi*`. Note que aqui não temos os

parênteses. Essas formas de escrever são equivalentes e retornam os mesmos documentos.

conteudo:(java AND cdi)

Recupera os documentos que contêm as palavras `java` e `cdi` em qualquer posição no conteúdo. Para consultas com mais de uma palavra é obrigatório utilizar os parênteses. É o mesmo que `conteudo:(cdi AND java)` ou `conteudo:cdi AND conteudo:java`.

> Observação: não é o mesmo que `conteudo:java cdi`, pois a palavra `cdi` não está entre parênteses. No nosso exemplo, a palavra `cdi` será desconsiderada, por isso a importância de não esquecer os parênteses.

conteudo:(java OR cdi)

Recupera os documentos que contêm as palavras `java` ou `cdi` no texto. O operador padrão é `OR`, então `conteudo:(java OR cdi)` e `conteudo:(java cdi)` retornam os mesmos documentos.

> Para mudar o operador padrão do `QueryParser` de `OR` para `AND` utilize a sintaxe `parser.setDefaultOperator(Operator.AND)`.

Uma vez definido o operador padrão, ele pode ser omitido em suas futuras consultas.

conteudo:"rede social"

Recupera documentos que contêm o termo exato `rede social` e nenhuma de suas variações. Portanto, não considera `redes sociais` ou `social rede`.

conteudo:monitor*

Recupera os documentos que contêm o prefixo `monitor` e as suas variações: `monitor`, `monitora`, `monitoramento`, `monitoração`, `monitorando` etc. Outro exemplo seria o prefixo `cart`, que pode representar `carteira`, `carteiro`, `carta`, `cartola` etc.

> Por padrão, o Lucene não permite consultar com o prefixo asterisco (`*`), como em `conteudo:*omitor` ou `conteudo:?onitor`.

Esse recurso, chamado de *leading wildcard*, pode resultar em perda de performance, porque o Lucene teria que varrer uma grande parte do índice para encontrar os documentos. Dependendo da quantidade de palavras encontradas, o consumo de memória da aplicação também pode ser um problema.

Lembra do índice invertido? Ele não é eficiente para resolver diretamente esse problema. Entretanto, há alternativas se você precisar executar esse tipo de consulta que serão vistas no capítulo *9. Recursos avançados*.

Exemplo: considere a consulta `conteudo:*mente`, com o objetivo de recuperar documentos que contêm os advérbios: `fracamente`, `bravamente`, `alternativamente` etc. Para usar esse recurso, você deve habilitá-lo explicitamente no *parser*, como visto na listagem a seguir.

```
parser.setAllowLeadingWildcard(true);
```

Quando executar uma consulta com *leading wildcard* sem habilitar o recurso, será lançada uma `ParseException` mostrando que os caracteres asterisco (`*`) e interrogação (`?`) não são permitidos como prefixo em uma consulta. A mensagem é `'*' or '?' not allowed as first character in WildcardQuery`.

conteudo:monitor?

Recupera apenas os documentos que contêm o prefixo `monitor` e mais uma letra, como em `monitora` e `monitore` e `monitoro`. O mesmo

princípio pode ser usado para outros prefixos, como `corrid?`, que se refere tanto a `corrida` quanto a `corrido`.

Os caracteres asterisco (`*`) e interrogação (`?`) são *wildcards*; são substitutos de outros caracteres. A diferença entre eles é que o asterisco substitui uma quantidade ilimitada de caracteres (zero ou vários), enquanto a interrogação substitui um e *apenas um* caractere.

conteudo:manuel~

Usamos o caractere til (`~`) para fazer buscar por termos imprecisos, ou *fuzzy query*. A ideia é recuperar documentos que contenham palavras semelhantes ao termo pesquisado. Ela utiliza a *distância de Levenshtein*, que é uma métrica para definir a diferença entre palavras. Para calcular essa diferença é usada *distância de edição*, que é a quantidade de edições para transformar uma palavra em outra, isto é, quantos caracteres precisariam ser trocados para converter uma palavra em outra.

Veja o caso da palavra `manuel`. Para transformar em `manual` precisamos trocar apenas 1 caractere (a -> e). Agora, vamos analisar a palavra `seção`. Com apenas 1 mudança podemos criar palavras como "serão" e "senão". Com 2 mudanças podemos criar várias alternativas como "seleção", "são", "ação", "estão", "sendo", "isenção" etc.

No exemplo a seguir, vamos pesquisar os parlamentares que se chamam `manuel` e outros nomes semelhantes como `Arthur`. A *fuzzy query* permite um parâmetro numérico que varia de 0 a 2, indicando a quantidade de edições para transformar um termo em outro. É possível usar `conteudo:manuel~0`, `conteudo:manuel~1` ou `conteudo:manuel~2`. Mais detalhes podem ser vistos aqui (https://pt.wikipedia.org/wiki/Dist%C3%A2ncia*Levenshtein/).

- `conteudo:wellington~` - recupera documentos com nome como wellington, welington;

- `conteudo:luis~1` - encontra luis, luís, louis, luiz;
- `conteudo:lucio~1` - encontra lucia, luci, lucio.

data:[20180101 TO 20180630]

Recupera os documentos com data de alteração entre `01-01-2018` e `30-06-2018`, incluídas essas datas. Esta é uma busca por intervalos, onde queremos encontrar os documentos indexados entre essas datas. Para encontrar os documentos de uma única data, use a consulta tradicional: `data:20180101`. Este assunto voltará mais tarde neste capítulo.

data:{20180101 TO 20180630}

Recupera os documentos com data de alteração entre 01-01-2018 e 30-06-2018, excluindo os documentos alterados nessas datas.

Você pode combinar os símbolos, usando chave e colchete, ou vice-versa. Lembre-se apenas de que o colchete significa que o valor será incluído no resultado, e com chave o valor não será incluído no resultado.

conteudo:"proposta reforma"~5

A busca por proximidade recupera os documentos nos quais os termos indicados estão separados por uma quantidade definida de palavras. É um filtro que indica se os termos estão relacionados, porque se estão próximos um do outro significa uma provável associação. No nosso exemplo, se os termos `proposta` e `reforma` estiverem com até 5 palavras de distância, provavelmente há uma relação.

Podemos concluir que palavras próximas estão relacionadas, enquanto palavras muito distantes no texto têm pouca ou nenhuma ligação. Imagine que você está planejando sua viagem para Paris e procura por `viagem para paris`. Se as palavras `viagem` e `paris` estiverem muito próximas no texto, provavelmente você quer

recuperar esse documento. Mas se estiverem muito separadas, por exemplo no início e no fim do texto, dificilmente esse documento fala sobre o assunto que nos interessa.

Observe que a ordem é importante. A busca é por `proposta -> reforma` e não `reforma -> proposta`. É isso que significa a sintaxe `conteudo:"proposta reforma"~5`. Estamos procuramos documentos que contêm as palavras `proposta` e `reforma` com até 5 palavras de distância.

Se os termos estão muito distantes, provavelmente não estão relacionados. Quando o termo `proposta` está no começo do texto e o outro termo (`reforma`) está no final, possivelmente não existe vínculo entre eles.

Considere os seguintes exemplos de texto:

1. *(...) proposta de reforma (...)*: os termos pesquisados têm 1 palavra de distância;
2. *(...) proposta que está em tramitação de reforma da previdência (...)*: neste caso, os termos estão a exatas 5 palavras de distância;
3. *(...) proposta atual não está de acordo com a anterior de reforma (...)*: os termos pesquisados estão a 9 palavras de distância;
4. *(...) a reforma é diferente da proposta anterior (...)*: os termos estão a 3 palavras de distância.

O texto do item 1 entra na nossa pesquisa. O item 2 também atende ao critério de consulta. O caso do item 3 não entra em nosso resultado porque os termos pesquisados estão a 9 palavras de distância e no último caso o texto está fora do nosso resultado porque os termos estão invertidos (reforma -> proposta).

Recupera todos os documentos (`*:*`)

Existe uma consulta que retorna todos os documentos do índice. Apenas pesquise por `*:*` (asterisco, dois pontos e asterisco) que o

buscador recuperará todos os itens indexados. Vale lembrar que o método `search(query, n)` limita a quantidade de itens aos *n* primeiros. Nosso buscador está limitado aos 100 primeiros itens. Se você pretende mostrar mais de 100 itens no resultado da consulta, aumente esse valor.

4.4 Operadores lógicos

As consultas até agora têm utilizado critérios simples, mas às vezes precisamos usar várias restrições agregadas com o objetivo de filtrar os resultados. Para isso, criamos consultas combinando vários campos usando operadores lógicos.

É importante perceber que os operadores lógicos devem ser escritos em letras maiúsculas. Os operadores são o `OR` , o `AND` , que pode ser substituído pelo caractere `+` (mais), e o `NOT` , que pode ser substituído pelo caractere `-` (hífen).

Os 3 exemplos a seguir são equivalentes e retornam o mesmo resultado. Observe que os 3 termos são obrigatórios porque usamos o operador `AND` ou o símbolo equivalente `+` . Atenção aos parênteses da última linha. O objetivo dessa consulta é encontrar todos os documentos com os termos `java` e `cdi` em seu conteúdo, que tenham sido modificados no ano de 2018.

- `conteudo:java AND conteudo:cdi AND data:[20180101 TO 20181231]`
- `+conteudo:java +conteudo:cdi +data:[20180101 TO 20181231]`
- `+conteudo:(+java +cdi) +data:[20180101 TO 20181231]`

As 3 consultas a seguir são equivalentes e utilizam o operador `NOT` ou o símbolo equivalente, `-` . Neste caso, o objetivo é recuperar documentos que tenham a palavra `java` mas que não tenham a palavra `cdi` no seu conteúdo e que tenham sido modificados em 2018. É importante notar os parênteses na última linha para garantir o agrupamento dos critérios.

- `conteudo:(java NOT cdi) AND data:[20180101 TO 20181231]`
- `+conteudo:(java -cdi) +data:[20180101 TO 20181231]`
- `+(conteudo:java -conteudo:cdi) +data:[20180101 TO 20181231]`

4.5 Busca com elevação (boost)

A ordem dos itens no resultado de uma consulta é definida através do mecanismo de pontuação, como visto na seção *Relevância* do capítulo *2. Conceitos de recuperação da informação*. É um cálculo matemático que define a classificação de cada item. É, basicamente, o coração do Lucene e sua eficiência tornou essa biblioteca tão popular.

Itens mais importantes, segundo esse cálculo, aparecem primeiro. Em alguns casos pode ser que a ordenação não atende ao usuário. Para isso, existe uma forma de alterar esses cálculos quando você quer dar ênfase a uma parte da consulta que é mais importante.

Vamos imaginar a consulta `conteudo:(nuvem rede)`, onde o usuário procura por documentos com as palavras `nuvem` ou `rede`. Ele quer saber sobre nuvens e redes de computadores, uma área nova e importante da computação.

> Uma observação: esse tópico não funciona muito bem com os termos `java` e `cdi` porque o segundo (`cdi`) tem provavelmente menos documentos e, assim, estará nas primeiras posições, com ou sem elevação. O *boost* faz mais sentido quando os termos tiverem a mesma relevância, como no caso de `nuvem` e `rede` ou quando a consulta envolve vários campos diferentes.

A elevação (*boost*) pode ser feita tanto no campo quanto no documento inteiro. Nesta seção veremos como é feita no campo. Mais detalhes estão descritos no próximo capítulo, *5. Principais*

classes. Por ora, o que precisamos saber é como e por que usar o recurso.

O *boost* não altera os itens do resultado da consulta, apenas sua ordenação. Nossa consulta inicial é `conteudo:(nuvem rede)`, onde cada campo tem elevação 1. Se considerar que o segundo termo (`rede`) é mais importante, você pode aumentar seu *boost* com a sintaxe rede^2, ou seja, o peso desse termo é maior que o peso primeiro termo. A consulta fica assim: `conteudo:(nuvem rede^2)`.

O resultado da busca contém exatamente os mesmos documentos, contudo, os itens que incluem a palavra `rede` estarão nas primeiras posições. Você pode usar qualquer número positivo de ponto flutuante como fator de elevação, inclusive números menores que 1, como 0.1 ou 0.9.

4.6 Expressão regular

Expressão regular (*regular expression*, regex ou regexp) é uma sequência de caracteres dentro de um texto que define um padrão. Alguns são facilmente reconhecidos e fazem parte do nosso cotidiano, como e-mail, telefone, data e hora. Para o pessoal de tecnologia há mais alguns padrões como HTML, XML e número IP (Internet Protocol).

Quando lemos um texto é fácil identificar um e-mail quando temos um termo assim: `ma@marcoreis.net` ou `jose@gmail.com`. Da mesma forma, um telefone é identificado como sendo uma sequência de 8 (ou 9) dígitos, com um separador no meio. Assim: 8119-4620, 98119-4620 ou até mesmo 9-8119-4620. O mesmo vale para as datas: 01/01/2018 é claramente uma data, bem como 01-01-2018. Veja que os separadores podem ser ou uma barra (`/`) ou um hífen (`-`). E ainda existem as datas em formato invertido, como 2018-01-01. Mas há restrições, pois não existe a data 30/02/2018. O padrão

para identificar uma hora também é simples. Uma hora é representada por algo como 18:00, 12:59 ou 8:00. Contudo, certamente não pode ser 12:60, nem 24:00.

Esses padrões são facilmente identificados por uma pessoa, mas não por um computador. Você pode buscar com o Lucene quais são os documentos que contêm o texto `ma@marcoreis.net` ou o texto `8119-4620` . Acontece que o buscador não sabe que isso é um e-mail e um telefone. Se quiser pesquisar os documentos que contêm e-mail ou número de telefone, tem de usar uma expressão regular para ensinar o padrão ao computador.

Vamos discutir um caso prático em um site de compras que tem a política de não permitir que o vendedor e o comprador troquem dados de contato. Isso faz muito sentido porque se as duas partes se conhecem não precisam pagar comissão para o site de compras. Você consegue até impedir que os envolvidos troquem insultos, porque é fácil identificar um palavrão. Mas não tem como impedir que o usuário digite seu e-mail ou telefone. Para evitar esse tipo de situação você precisa de um buscador com suporte a expressão regular.

E as opções de busca aumentam quando utilizamos regex, entretanto, aumentam igualmente as complicações. Não tem como obrigar o usuário a digitar, por exemplo, a hora com formato correto, supondo que utilizamos o formato hh:mm. Não podemos impedir que o usuário digite com outro formato, como 18h, 18hr, 18hrs, 18 horas e por aí vai. A data pode usar o formato brasileiro, o formato americano (yyyyMMdd). Mas com certeza alguém vai escrever uma data no formato *dia 11 de dezembro*.

Outros padrões são menos triviais. É o caso dos sistemas de busca que usam vocabulário especializado, como Matemática, Medicina e Direito. Vamos considerar um buscador especializado para advogados onde o usuário procura por uma legislação em particular dentro de uma coleção de documentos jurídicos.

Exemplo: Lei 8.112, Lei 8.666 de 1993, Art. 138 do Código Penal ou Art. 5 do CPC. Note que há um padrão nesses casos. Podemos ter um termo como `Lei` ou `Art.` seguindo de um número e, opcionalmente, uma data ou a descrição do código. Esses casos têm de ser tratados se você precisa de uma grande precisão. Ainda assim, um ou outro documento acaba ficando de fora da nossa busca porque o usuário digitou muito fora do esperado.

O Google (e qualquer outro buscador) continuamente faz análises para identificar se as buscas dos usuários estão sendo efetivas. Os motores de busca são refinados para conseguir encontrar os documentos com a melhor exatidão possível naquele momento. A linguagem é dinâmica e a forma de escrever muda com frequência. Faz parte dos sistemas de busca acompanhar essa evolução, uma vez que a base de tudo é a linguagem natural.

Como visto, são muitas situações onde o uso de regex é interessante. No caso dos sistemas de busca eu diria que é extremamente útil conhecer regex. Mas também é um conhecimento importante para programadores em geral. Neste capítulo veremos os principais padrões e algumas dicas para encontrar elementos específicos no texto.

Um bom ponto inicial é o site (http://www.regular-expressions.info/). Apesar de estar em inglês o conteúdo é acessível e bem organizado. Se souber o que está procurando, é possível encontrar a expressão regular e usar. No capítulo *9. Recursos avançados*, serão mostrados outros padrões e expressões mais complexas, que não cabem no conteúdo visto até agora.

Expressões regulares no Lucene

Muitas ferramentas implementam regex, incluindo as linguagens de programação, editores de texto e, no Linux/Unix, temos o SED e o AWK, além das ferramentas específicas para processamento de texto. Então, para trabalhar com processamento de texto certamente aplicaremos muitas expressões regulares.

O Lucene tem suporte para expressões regulares e, de uma forma bem genérica, pode-se dizer que é possível usar todos os recursos de busca apenas com regex. Desde que você saiba bem como criar a expressão correta. Para consultar com expressão regular usamos a sintaxe `campo:/expressão-regular/` . Observe que a expressão regular fica entre barras (`/`). Os padrões analisados consideram sempre o texto completo.

Outra fonte de informação sobre o assunto é a documentação da classe `RegExp` do Lucene (vide capítulo de referências do livro), onde está especificado o suporte para expressão regular. Ele obedece a essa sintaxe:

- `|` (pipe): união de valores;
- `&` (ampersand): interseção de dois padrões;
- `?` (interrogação): zero ou uma ocorrência;
- `*` (asterisco): zero ou mais ocorrências;
- `+` (mais): uma ou mais ocorrências;
- `{n}` : n ocorrências;
- `{n,}` : n ou mais ocorrências;
- `{n,m}` : n para m ocorrências, incluindo ambas;
- `~` (til): complemento;
- `[^*string*]` : não ocorrência (negação) da *string*;
- `^regex` : a regex ocorre no início do texto;
- `regex$` : a regex ocorre no fim do texto;
- `-` (hífen): intervalo de caracteres, incluindo ambos;
- `.` (ponto): uma ocorrência de qualquer caractere;
- `#` (cerquilha): vazio;
- `@` (arroba): qualquer *string*. É equivalente a ponto asterisco (`.*`);
- `"alguma *string* UNICODE"` : uma *string* qualquer;
- `()` : uma *string* vazia;
- `(a|b)` : define precedência;
- `<n-m>` : intervalo numérico;
- `\algum caractere UNICODE` : ocorrência de um caractere;
- `[caracteres]` : qualquer um dos caracteres.

A seguir estão listadas algumas formas de usar esses caracteres:

- `/[cb]arro/` : carro ou barro;
- `/.arro/` : carro, barro, sarro etc;
- `/<.+>/` : encontra uma tag HTML;
- `/[a-e1-5]/` : intervalo das letras `a` até `e` e dos dígitos 1 até 5;
- `/[abc]/` : qualquer um dos caracteres, ou `a` ou `b` ou `c` ;
- `/[^abc]/` : qualquer caractere que não seja `a` ou `b` ou `c` ;
- `/[F-H]/` : intervalo das letras maiúsculas `F` até `H` ;
- `/cachorro|gato/` : combina com cachorro ou gato.

Agora, considere a *string abcde* para os padrões a seguir.

- `/ab@/` : combina;
- `/ab.*/` : combina;
- `/abcd/` : não combina;
- `/ab.../` : combina;
- `/ab.../` : combina;
- `/a.c.e/` : combina.

Dessa vez considere a *string* "aaabbb".

- `/a+b+/` : combina;
- `/a*b*/` : combina;
- `/aaa?bbb?/` : combina;
- `/a+.+/` : combina;
- `/a*b*c*/` : combina;
- `/a*b*c+/` : não combina;
- `/.*bbb.*/` : combina;
- `/aaaa?bbbb?/` : combina;
- `/a{3}b{3}/` : combina;
- `/a{2,}b{2,}/` : combina;
- `/aaa(ccc|bbb)/` : combina;
- `/aa~bb/` : combina;
- `/aaa.+&.+bbb/` : combina;
- `/aaa&bbb/` : não combina.

Alguns caracteres são especiais para as expressões regulares e têm outro significado neste contexto, como é o caso do hífen. Eles são reservados para controle da regex e, se precisar usá-los, você tem que escapá-los com uma contra-barra (`\`). Por esse motivo usamos uma contra-barras com o hífen, porque precisamos do valor literal de um caractere de controle. A lista de caracteres reservados é:

- `: acento grave;
- `.` : ponto;
- `?` : interrogação;
- `+` : soma;
- `|` : barra vertical;
- `{}` : abertura e fechamento de chaves;
- `[]` : abertura e fechamento de colchetes;
- `()` : abertura e fechamento de parêntesis;
- `"` : aspas duplas;
- `\` : barra invertida;
- `#` : *hashtag*;
- `@` : arroba;
- `&` : ampersand ou *e comercial*;
- `<>` : sinal maior e menor;
- `~` : til.

Vamos consultar alguns casos envolvendo dígitos. Considere a *string* abcdef100.

- `/abcdef<1-100>/` : combina;
- `/abcdef<1-99>/` : não combina.

Começaremos a parte prática com uma busca por uma sequência de 4 dígitos, ou seja, um valor numérico entre 0 e 9. O objetivo é recuperar documentos que contenham em seu conteúdo no mínimo 4 dígitos quaisquer. Por exemplo: 1234, 0123 ou 0000. A expressão regular para esse padrão é `[0-9][0-9][0-9][0-9]` ou, de forma reduzida, `[0-9]{4}` .

Como o conteúdo dos nossos arquivos indexados não tem apenas o número, precisamos considerar que os documentos contêm outras palavras e a sequência numérica fica no meio de tudo. Neste caso, adicionamos o critério *ponto asterisco* (`.*`) antes e depois da expressão regular. Esse critério diz para o analisador do regex considerar qualquer caractere antes e depois do número.

No exemplo a seguir temos um texto aleatório onde aparece o número `9876` bem no meio. Como há caracteres antes e depois, a regex `[0-9]{4}` não combina com nosso texto. O correto para este caso seria `.*[0-9]{4}.*` . Você pode ler isso como sendo: um texto que inicia com zero ou vários caracteres, uma sequência de 4 dígitos e finaliza com zero ou vários caracteres. Veja o texto:

```
Lorem ipsum dolor sit amet, consectetur adipiscing elit. Fusce lobortis sodales arcu mollis interdum. Sed a metus 9876 dictum, hendrerit risus quis, eleifend lacus. Nullam vitae fermentum eros.
```

Caso o documento tenha, em seu conteúdo, uma sequência de 3 números, não será considerado porque indicamos uma quantidade mínima. Contudo, uma sequência de 5 números fará parte do resultado da busca. Em uma regex é importante indicar em que posição do texto está seu objetivo. Podemos indicar que nosso objetivo está no começo, no fim ou em qualquer parte, como no nosso caso.

Para usar expressões regulares no Lucene precisamos de um novo campo indexado. Será o campo `conteudoNaoAnalisado` e, como o nome sugere, seu conteúdo do texto não será analisado pelo motor de busca. Com isso, o `Analyzer` não fará nenhum pré-processamento, como a eliminação de *stop words* e de caracteres especiais.

Por que precisamos disso? Porque o analisador do Lucene altera o conteúdo do texto quando usamos um campo do tipo `TextField` . Isso significa que o campo `conteudo` não é exatamente igual ao texto extraído do arquivo original. Sendo assim, as expressões regulares

que você usar podem não funcionar, o que é verdade principalmente para campos numéricos.

> Para usar expressão regular, use um campo indexado como `StringField` e não `TextField`.

Esta é a alteração que deve ser feita na classe `IndexadorArquivosLocais`. Adicione a linha `doc.add(new StringField("conteudoNaoAnalisado", textoArquivo, Store.YES));` no método `indexarArquivo`, como no código a seguir. Depois de alterar o código e indexar novamente os arquivos, teremos o campo `conteudoNaoAnalisado` com o conteúdo original do arquivo.

A opção `Store.YES` indica que o conteúdo deste campo será armazenado. Para economizar espaço em disco, você pode usar `Store.NO` porque já temos outro campo que armazena esse conteúdo. O campo não pode ter mais de 32766 bytes, que é o limite para campos não analisados. É por isso que estamos limitando o tamanho para os primeiros 30000 bytes do texto.

```
// {...}
int tamanhoMaximo = 30000;
if (textoArquivo.length() >= tamanhoMaximo) {
    doc.add(new StringField(
            "conteudoNaoAnalisado",
            textoArquivo.substring(0,
                    tamanhoMaximo),
            Store.YES));
} else {
    doc.add(new StringField(
            "conteudoNaoAnalisado",
            textoArquivo, Store.YES));
}
// {...}
```

Na classe `BuscadorSintaxeClassicaTest`, vamos adicionar o método `testConsultaRegex`. Ele realiza uma busca usando a expressão

regular `/.*[0-9]{4}.*/` , ou seja, uma sequência de 4 dígitos no meio de um texto. Execute o teste e vamos analisar o resultado. O esperado é que este método recupere os documentos que têm uma sequência de 4 dígitos.

```
public void testConsultaRegex() {
    BuscadorArquivosLocais buscador =
            new BuscadorArquivosLocais();
    String regex = "/.*[0-9]{4}.*/";
    String consulta = "conteudoNaoAnalisado:" + regex;
    buscador.buscar(consulta);
}
```

Regex para número de telefones

Definimos o padrão para o número de telefone como sendo uma sequência de 8 ou 9 números, separados ou não por hífen. O padrão seria `dddd-dddd` , onde `d` é um dígito. Claro que é um padrão inicial e depois faremos algumas melhorias. A expressão regular para isso é `[0-9]{4}\-\[0-9]{4}` . São 3 parâmetros:

- `\[0-9]{4}` : uma sequência de 4 dígitos;
- `\-` : um caractere hífen;
- `\[0-9]{4}` : outra sequência de 4 dígitos.

O resultado trouxe (ou deveria ter trazido) os documentos que contêm números de telefone. Acontece que essa regex não é a mais correta para trazer números de telefone. Ela recupera números como 0123-4567. É um bom começo, mas ainda podemos melhorar.

Aqui em Brasília, por exemplo, os números de telefone não começam com 0 e 1, mas nossa primeira versão da regex não trata essa situação. Atualizando a expressão para esse caso, temos `[2-9][0-9]{3}\\-[0-9]` . Na nova expressão, o primeiro dígito precisa estar entre 2 e 9. Depois temos 3 dígitos, o hífen e mais 4 dígitos. Assim não recupera mais documentos que começam com 0 e 1. Você vai descobrir que ainda há exceções, como os 0800 e 0900. Fica como trabalho de casa fazer a adaptação.

Regex para registros de bugs

Digamos que você precisa encontrar documentos que falem sobre um *bug*, por exemplo, sobre o *bug* 1234. A expressão regular para isso é simples: `/.*bug.{1,5}[0-9]{4}@/` . Deixei os dois símbolos para *string* (@ e `.*`) para verificar que são equivalentes. Vamos analisar os elementos desse padrão:

- `.*` : uma sequência de caracteres quaisquer;
- `bug` : a palavra *bug*;
- `.{1,5}` : uma sequência de 1 a 5 caracteres;
- `[0-9]{4}` : uma sequência de 4 dígitos;
- `@` : outra sequência de caracteres quaisquer.

Regex para e-mail

Uma expressão regular útil é essa para descobrir se existe um e-mail no meio de um texto. `/@[a-z0-9\\.\\*\\%\\+\\-]+\\@[a-z0-9\\.\\-]+\\.[a-z]{2,}@/` . Aqui estamos usando arroba (@) no lugar de ponto asterisco (`.*`) para mostrar que o efeito é equivalente, ou seja, que o e-mail pode estar em qualquer posição do texto. Os elementos do nosso e-mail são esses:

- `[a-z0-9\\.\\*\\%\\+\\-]+` : a primeira parte do e-mail pode ser uma sequência de 0 ou mais caracteres, permitindo os símbolos `*. * % + -*`
- `\\@` : o símbolo escapado arroba
- `[a-z0-9\\.\\-]+` : este é o domínio do e-mail, que permite apenas letras, números e os símbolos `._-`
- `\\.` : o símbolo do ponto escapado
- `[a-z]{2,}` : um conjunto de 2 ou mais caracteres para finalizar. Este é o *top-level domain* e pode ser .com, net, .br etc.

Parece estranho, mas é possível sim existir os símbolos `% +` na conta de e-mail. É comum em artigos científicos para agrupar os endereços.

Regex para número IP

Um número IP (*Internet Protocol*) é representado por uma sequência de 4 dígitos separados por ponto (0.0.0.0) como o famoso *loop back* (127.0.0.1). O padrão para encontrar números IP é `"/@[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}\\.[0-9]{1,3}@/"` . Imagine que podemos usar o endereço IP para substituir uma URL. O IP 184.107.114.5, por exemplo, redireciona para www.marcoreis.net. Este é um padrão simples, com a repetição dos dígitos `[0-9]{1,3}` 4 vezes.

Regex para data

Para encontrar documentos com datas no formato brasileiro (dd/MM/yyyy), usamos a expressão `/@(0[1-9]|[1-2][0-9]|3[01])[\\- \\/\\.](0[1-9]|1[012])[\\- \\/\\.](19|20)[0-9][0-9]@/` . Esta é um pouco mais complicada, mas quando separamos as partes o entendimento fica mais simples. Veja:

- `(0[1-9]|[1-2][0-9]|3[01])` : encontra o dia, que pode ser 01, 11 ou 31. Essas 3 opções são definidas pelo *pipe* (`|`).
- `[\\- \\/\\.]` : identifica os separadores, que podem ser hífen, espaço, barra e ponto.
- `(0[1-9]|1[012])` : é o mês, que pode ser 02 ou 12. Novamente temos o *pipe* para alternar os padrões.
- `[\\- \\/\\.]` : separador.
- `(19|20)[0-9][0-9]` : consideramos o ano a partir de 1900, com isso, a menor data deste padrão é 01/01/1900.

Regex para intervalos numéricos

O Lucene suporta busca por intervalo numérico através dos símbolos `<valor1-valor2>` . Para encontrar os documentos com tamanho entre 0 e 500000, use a expressão `tamanho:/<0-500000>/"` . O campo `tamanho` tem apenas o número. Se quiser encontrar os documentos que contêm um intervalo de valores em qualquer parte do texto, use a sintaxe `conteudo:/@<0-500000>@/` .

Outras opções de regex

A partir de agora você vai encontrar padrões em tudo e com isso damos uma nova e interessante opção de consulta para nosso usuário, que pode pesquisar não apenas por valores literais, mas também através de expressões regulares. O próximo passo é criar eventos acionáveis a partir deles, como substituir valores e enviar notificações através de alertas.

Exemplo: um sistema de *e-commerce* pode permitir que o usuário cadastre alertas para situações específicas, como quando aparecer o produto desejado com preço determinado. São duas consultas agrupadas: uma pelo nome do produto, outra com expressão regular para intervalo de valores.

Além destes exemplos citados, outros padrões comuns em textos que encontramos com regex são:

- Documentação, como CPF, CNPJ e Passaporte;
- Linhas ou itens duplicados;
- Código-fonte;
- Nomes de arquivos;
- *Log* de aplicações;
- Código de rastreamento dos Correios;
- Anúncios de produtos e serviços;
- Valores monetários e percentuais;
- Cabeçalho e rodapé de documentos;
- Códigos de confirmação;
- Processos de tribunais.

Estes padrões serão muito utilizados no capítulo *10. Extraindo dados da internet*, onde faremos o rastreamento do conteúdo de sites com robôs web e, com a ajuda de expressões regulares, selecionaremos apenas os itens desejados. Essa técnica pode ser aplicada em conteúdo de notícias, viagens, empregos, PDFs etc.

Chegamos ao final da primeira parte do capítulo, onde falamos da sintaxe clássica do Lucene e como combinar suas opções para criar novas funcionalidades de busca para uma aplicação. Na próxima seção vamos rever as mesmas consultas usando a API do Lucene.

4.7 API do Lucene

Além da sintaxe clássica, podemos usar a API do Lucene para realizar buscas, o que será descrito nesta seção. Esta API contém um conjunto de classes e interfaces e estão no pacote `org.apache.lucene.search`, onde você encontrará opção para os mais diversos tipos de consulta. Em relação à performance, tanto a sintaxe clássica quanto a API são similares. Em termos de funcionalidade também são similares, porque é possível realizar o mesmo tipo de consulta com as duas. Com isso, podemos desenvolver nossa solução usando a sintaxe clássica, a API ou ambas. Não faz mesmo diferença no resultado. Dessa forma, nesta seção será mostrado como usar a API do Lucene para construir as buscas no índice. Basicamente, os exemplos são os mesmos da seção anterior, mas construídos a partir das classes disponíveis na API do Lucene.

Uma consideração a ser feita é que a consulta que utiliza a sintaxe clássica não tem validação na compilação. O programador escreve as consultas, como visto na seção anterior, e a validação é feita apenas durante a execução. Se a sua consulta tiver algum erro de sintaxe, o compilador Java não fará nenhuma crítica. Apenas quando o usuário acessar aquele recurso é que descobriremos o problema. Claro que um bom plano de testes resolve essa situação. No caso da API, como temos de criar objetos, seu compilador fará uma validação, mesmo que bastante básica.

Outro ponto é a complexidade da sua solução. Para consultas simples, com poucos critérios, a sintaxe clássica pode ser uma boa

escolha. Para soluções mais complexas, por exemplo, a construção de critérios dinâmicos de uma busca avançada, a API pode ser mais interessante porque o código-fonte ficará mais legível. É o caso de uma consulta avançada onde o usuário pode escolher entre vários critérios diferentes na tela de consulta. Com a API, pode-se combinar diversos objetos e criar uma busca complexa com tudo. No caso da sintaxe clássica, uma grande String de consulta pode ser confusa.

Antes de começar o trabalho, vamos relembrar quais as principais opções de busca, vistas no decorrer do livro:

- Palavra-chave: tipo mais básico de busca, encontra os documentos que contêm aquela palavra pesquisada.
- Por frases: similar ao anterior, encontra documentos com todas ou qualquer uma das palavras indicadas.
- Buscas booleanas, com a combinação de outras buscas e operadores lógicos (AND, NOT e OR).
- Com elevação (*boost*) de termo ou de campo: tipo de busca que eleva o peso que um termo ou campo tem na classificação do resultado. Por exemplo, campos como categoria ou título geralmente têm um peso maior na busca.
- Busca difusa (*fuzzy*): recupera os documentos que contêm variações da palavra pesquisada. É o caso do usuário que digita errado alguns caracteres. Exemplo: capcioso, vicissitudes e procrastinar.
- Prefixada ou com caractere curinga: encontra os itens considerando um prefixo ou outras partes da palavra.
- Por proximidade dos termos: seleciona documentos onde as palavras pesquisadas estão perto uma da outra.
- *More like this*: encontra documentos semelhantes. Por exemplo: dado um documento qualquer, essa busca encontra outros documentos semelhantes. Está em um pacote diferente, o `org.apache.lucene.queries` .
- E uma das minhas preferidas, que é a busca com expressão regular: encontra documentos através de expressões regulares.

Útil para encontrar documentos com e-mail, telefones ou padrões de frases.

A classe BuscadorAPITest

Como já foi visto, nosso trabalho será refazer as consultas anteriores, mas desta vez com a API no lugar da linguagem de consulta clássica. O resultado retornado deve ser idêntico. A classe criada para a tarefa se chama `BuscadorAPITest`, e seu código está a seguir:

```
public class BuscadorAPITest {
    private static final Logger logger =
        Logger.getLogger(BuscadorAPITest.class);
    // {...}
}
```

Mais uma vez usamos a classe `BuscadorArquivosLocais`, vista no capítulo anterior. Acrescentei o método `public void buscar(Query query)`. A diferença é que neste método não usamos o `QueryParser`, uma vez que ele faz parte da sintaxe clássica. Aqui usaremos apenas a `Query` e suas subclasses, que serão explicadas logo a seguir, começando com `TermQuery`.

TermQuery

Para cada tipo de busca existe uma classe na API. A busca mais simples é, naturalmente, por palavra-chave. Esta é a `TermQuery` e a seguir temos um exemplo de como a usar. Este modelo retorna o mesmo resultado que a primeira consulta com a sintaxe clássica `conteudo:java`. Neste caso criamos um objeto `Term` e um `TermQuery`, como na listagem de código a seguir. Este bloco deve ser colocado na `BuscadorAPITest`, assim como os demais na sequência. O importante a observar é que o buscador recebe um objeto do tipo `Query` como parâmetro.

```
    public void testTermQuery() {
        logger.info("Consulta TermQuery");
```

```
        BuscadorArquivosLocais buscador =
                new BuscadorArquivosLocais();
        Term term = new Term("conteudo", "java");
        Query query = new TermQuery(term);
        buscador.buscar(query);
    }
```

MatchAllDocsQuery

A `MatchAllDocsQuery` é uma consulta sem parâmetros que retorna todos os documentos do índice. Como pode ser visto na listagem, não tem nenhum parâmetro e apenas retorna uma subclasse de `Query`. Nem precisamos dizer que esse recurso pode resultar em um problema de performance, considerando que estamos recuperando todos os documentos do índice. Para índices pequenos não é problema, mas para índices grandes, com bilhões de itens, podem ocorrer complicações de performance. Veja o código:

```
    public void testMatchAllDocsQuery() {
        BuscadorArquivosLocais buscador =
                new BuscadorArquivosLocais();
        MatchAllDocsQuery query = new MatchAllDocsQuery();
        buscador.buscar(query);
    }
```

BooleanQuery

Para combinar múltiplos critérios usamos a `BooleanQuery`. Na sequência, a próxima que vamos trabalhar é `conteudo:(java cdi)`, onde queremos todos os documentos que contêm qualquer um dos termos. Na verdade, esta é uma consulta com o operador lógico `OR`, que é o padrão. A classe da API é a `BooleanQuery`, que inclui uma classe estática `Builder` utilizada para adicionar os elementos da nossa consulta booleana.

A `BooleanQuery` permite a busca de documentos através da combinação de várias consultas. Permite a criação de consultas

com a combinação de até 1024 cláusulas e podem ser usados todos os tipos de consultas disponíveis.

Note que existe um parâmetro `Occur` para sinalizar qual o operador lógico queremos usar com aquele termo. São estas as opções:

- `MUST` : o termo deve aparecer no documento;
- `FILTER` : o termo deve aparecer no documento, mas não é considerado no cálculo da ordenação;
- `SHOULD` : o termo pode ou não ocorrer no documento;
- `MUST_NOT` : o termo não pode ocorrer no documento.

Se houver apenas uma cláusula na `BooleanQuery` , a consulta vai retornar os documentos que atendem a esse parâmetro, mesmo com a ocorrência `SHOULD` . Caso tenha várias cláusulas com `SHOULD` , a consulta retornará os documentos que contêm qualquer um dos parâmetros. Nesse caso, funcionará com o operador lógico `OU` . Se a `BooleanQuery` tem mais de uma cláusula com ao menos uma ocorrência `MUST` , a consulta vai retornar os documentos que atendem à cláusula `MUST` e vai utilizar as cláusulas `SHOULD` para calcular a pontuação do item, ou seja, o `SHOULD` vai indicar a ordem em que o documento vai aparecer no resultado.

```
public void testBooleanQueryShould() {
    logger.info("Consulta BooleanQuery");
    BuscadorArquivosLocais buscador =
            new BuscadorArquivosLocais();
    Query q1 = new TermQuery(new Term("conteudo", "java"));
    Query q2 = new TermQuery(new Term("conteudo", "cdi"));
    BooleanQuery query = new BooleanQuery.Builder()
            .add(q1, Occur.SHOULD).add(q2, Occur.SHOULD).build();
    buscador.buscar(query);
}
```

Para usar o operador booleano `AND` , o correspondente na `BooleanQuery` é o `Occur.MUST` . É o caso da consulta `conteudo:(java AND cdi)` , onde queremos que os dois termos ocorram obrigatoriamente nos documentos. Veja o código:

```
Query q1 = new TermQuery(new Term("conteudo", "java"));
Query q2 = new TermQuery(new Term("conteudo", "cdi"));
BooleanQuery query = new BooleanQuery.Builder()
        .add(q1, Occur.MUST).add(q2, Occur.MUST).build();
```

O outro operador que resta é o `NOT`, com seu correspondente `Occur.MUST_NOT`. A consulta `conteudo:(java -cdi)`, onde queremos documentos com a palavra `java` e que não tenham de forma alguma a palavra `cdi`, fica assim com a API:

```
Query q1 = new TermQuery(new Term("conteudo", "java"));
Query q2 = new TermQuery(new Term("conteudo", "cdi"));
BooleanQuery query = new BooleanQuery.Builder()
        .add(q1, Occur.MUST).add(q2, Occur.MUST_NOT).build();
```

Estamos aqui trabalhando com conjuntos de dados. A primeira consulta, com o operador `SHOULD`, é o maior conjunto porque tem documentos com `java` ou `cdi`. Vamos chamar de conjunto 1. A segunda consulta com o `MUST` é a mais restrita, porque os documentos precisam conter os dois termos. É um subconjunto da primeira, vamos chamá-la de subconjunto 1. A última consulta é outro subconjunto do conjunto 1. Somando os subconjuntos 1 e 2, temos de volta o conjunto 1.

Agora, vamos combinar múltiplos campos com a `BooleanQuery`. O exemplo a seguir mostra uma busca com os campos `conteudo` e `data`. Note que estamos usando o `Occur.MUST`.

```
Query q1 = new TermQuery(new Term("conteudo", "java"));
Query q2 = new TermQuery(new Term("data", "20170523"));
BooleanQuery query =
        new BooleanQuery.Builder().add(q1, Occur.MUST)
                .add(q2, Occur.MUST).build();
```

PhraseQuery

Para consultar frases, temos a classe `PhraseQuery`. É o caso da consulta exata `conteudo:"rede social"`, que pode ser obtido com a sintaxe:

```
Query query = new PhraseQuery("conteudo", "rede",
        "social");
```

A `PhraseQuery` tem uma opção para buscar documentos quando seus termos estão a uma distância conhecida, chamada de *slop*. O exemplo de código a seguir encontra documentos que têm as palavras `proposta` e `reforma`, nesta ordem, desde que estejam a até 5 palavras de distância.

```
Query query = new PhraseQuery(5, "conteudo", "proposta",
        "reforma");
```

SpanQuery, SpamTermQuery e SpanNearQuery

Mas e se você não quer considerar a ordem dos termos? Aí você usa uma `SpanQuery`. Ela tem uma opção para que você ignore a ordem em que as palavras aparecem no texto. Para tanto, use a sintaxe a seguir, onde estamos dizendo que a ordem não é importante, ou seja, os termos `proposta` e `reforma` podem aparece em qualquer ordem no documento.

O método construtor tem 3 parâmetros: os termos (`spans`), a quantidade de palavras de distância (`slop`) e se a ordem dos termos é importante ou não.

```
Term termoReforma = new Term("conteudo", "proposta");
Term termoProposta = new Term("conteudo", "reforma");
SpanQuery queryReforma = new SpanTermQuery(termoReforma);
SpanQuery queryProposta = new SpanTermQuery(termoProposta);
SpanQuery[] clausulas = new SpanQuery[] { queryProposta,
        queryReforma };
SpanNearQuery query = new SpanNearQuery(clausulas, 5, false);
```

Se quiser voltar a considerar a ordem dos termos, use a sintaxe

```
SpanNearQuery query = new SpanNearQuery(clausulas, 5, true) .
```

TermRangeQuery

A consulta por intervalos de valores utiliza a classe `TermRangeQuery`. É o caso das consultas `data:[20180101 TO 20181231]`, `data:[2018 TO 2017]`, `data:{20180101 TO 20181231}` e `conteudo:[ana TO beatriz]`. Nestas situações buscamos os documentos com valores dentro de um período de tempo ou faixa de valores.

Devemos preencher cinco parâmetros, todos obrigatórios: (i) o nome campo a ser pesquisado; (ii) o valor de limite inferior; (iii) o valor de limite superior; (iv) uma *flag* indicando se deve incluir o limite inferior; e (v) uma *flag* indicando se deve incluir o limite superior.

Os parâmetros estão separados em variáveis para tornar o entendimento mais simples. Nosso exemplo recupera todos os documentos que foram alterados em 2018, incluindo as datas de limite inferior e superior. Veja como ficou:

```
boolean incluirLimiteInferior = true;
boolean incluirLimiteSuperior = true;
BytesRef limiteInferior = new BytesRef("20180101");
BytesRef limiteSuperior = new BytesRef("20181231");
Query query = new TermRangeQuery("data", limiteInferior,
        limiteSuperior, incluirLimiteInferior,
        incluirLimiteSuperior);
```

Para excluir os documentos que estão na data limite, altere as *flags* como indicado a seguir.

```
boolean incluirLimiteInferior = false;
boolean incluirLimiteSuperior = false;
```

É possível combinar consultas diferentes, não apenas `TermQuery`. No próximo exemplo vamos usar duas consultas diferentes e combiná-las com uma `BooleanQuery`. O objetivo é encontrar os documentos com conteúdo `java` e que foram alterados em 2018.

```
BuscadorArquivosLocais buscador =
        new BuscadorArquivosLocais();
Query q1 = new TermQuery(new Term("conteudo", "java"));
boolean incluirLimiteInferior = true;
boolean incluirLimiteSuperior = true;
```

```
BytesRef limiteInferior = new BytesRef("20180101");
BytesRef limiteSuperior = new BytesRef("20181231");
Query q2 = new TermRangeQuery("data", limiteInferior,
        limiteSuperior, incluirLimiteInferior,
        incluirLimiteSuperior);
BooleanQuery query = new BooleanQuery.Builder()
        .add(q1, Occur.MUST).add(q2, Occur.MUST).build();
buscador.buscar(query);
```

De forma alternativa, podemos usar o método estático `TermRangeQuery.newStringRange` para o mesmo resultado:

```
query = TermRangeQuery.newStringRange("data", "20180101",
        "20181231", incluirLimiteInferior,
        incluirLimiteSuperior);
```

MultiPhraseQuery

Para criar uma `PhraseQuery` mais complexa, é preciso usar a `MultiPhraseQuery`. Ela também procura por documentos com uma sequência específica de termos, contudo, tem um método que permite adicionar diretamente uma lista de palavras nos critérios de busca, que é o `add(Term[])`, no qual você indica a lista de termos que compõem sua frase.

Vimos um exemplo simples de frase, como `conteudo:"rede social"`, onde encontramos os documentos que têm exatamente essa frase em seu conteúdo. Também vimos `conteudo:"proposta reforma"~5`, que encontra os documentos com essas palavras, nessa sequência, desde que estejam a até 5 palavras de distância. Mas e se quisermos combinar várias palavras?

A solução é usar a `MultiPhraseQuery`. No exemplo a seguir estamos buscando por documentos onde as palavras `java` e `platform` estão a até 5 palavras de distância dos termos `cdi` e `weld`. Você pode imaginar a `MultiPhraseQuery` como uma sequência de consultas, todas com o operador `OR`. Contudo, os dois grupos devem ocorrer no documento. Seria algo como `(java OR platform) AND (cdi OR weld)`.

Utilizamos o `slop` 5, então esses grupos devem estar a até 5 palavras de distância. Veja como ficou:

```
public void testMultiPhraseQuery() {
    BuscadorArquivosLocais buscador =
            new BuscadorArquivosLocais();
    Term[] termoJavaPlatform = new Term[] {
            new Term("conteudo", "java"),
            new Term("conteudo", "platform") };
    Term[] termoCdiWeld = new Term[] {
            new Term("conteudo", "cdi"),
            new Term("conteudo", "weld") };
    Query query = new MultiPhraseQuery.Builder()
            .add(termoJavaPlatform).add(termoCdiWeld)
            .setSlop(5).build();
    buscador.buscar(query);
}
```

Alguns programadores preferem separar as classes em várias linhas. No bloco de código a seguir temos a mesma consulta escrita em várias linhas para melhorar a leitura. O resultado é o mesmo, apenas está escrito de forma diferente.

```
Builder builder = new MultiPhraseQuery.Builder();
builder.add(termoJavaPlatform);
builder.add(termoCdiWeld);
builder.setSlop(5);
Query query = builder.build();
```

WildcardQuery

Um caractere curinga, ou *wild card*, é um símbolo que substitui uma sequência de caracteres desconhecidos. No Lucene, o asterisco (`*`) correspondente a uma sequência de 0 ou vários caracteres, enquanto a interrogação (`?`) corresponde a um único caractere.

Uma consulta que começa com o caractere coringa (`*` ou `?`) apresenta baixa performance se comparada às outras porque o Lucene precisa varrer uma grande área do índice para encontrar todos os documentos correspondentes.

O primeiro exemplo, com `monitor`, encontra todas os documentos com o prefixo `monitor` (inclusive), como `monitores` e `monitoramento`.

```
Term termo = new Term("conteudo", "monitor*");
Query query = new WildcardQuery(termo);
```

O próximo bloco de código, com `monitor?`, encontra os documentos com o prefixo `monitor` e mais um caractere, como em `monitora` e `monitore`. No entanto, não recupera `monitoria`, que tem 2 caracteres além do prefixo. Nesse caso, você pode pesquisar por `monitor??` (note que há 2 interrogações) para recuperar `monitoria`.

```
Term termo = new Term("conteudo", "monitor?");
Query query = new WildcardQuery(termo);
```

PrefixQuery

A `PrefixQuery` consulta documentos com base em um prefixo. O exemplo adiante vai consultar o prefixo `monitor` e suas variações como `monitoramento`, `monitorar` e `monitoria`. É equivalente a uma busca por `conteudo:monitor`.

```
Term termo = new Term("conteudo", "monitor");
Query query = new PrefixQuery(termo);
```

BoostQuery

A `BoostQuery` faz a elevação de um termo, ou seja, aumenta o peso deste termo durante a ordenação do resultado da consulta. O código a seguir é equivalente à consulta `conteudo:(nuvem rede^2)`, vista antes aqui no capítulo. Temos uma `BooleanQuery` com 2 termos, sendo que o segundo está elevado.

O ponto central aqui é a linha `BoostQuery boostQuery = new BoostQuery(queryRede, 2)`, que aumenta o peso do termo `rede` em nossa consulta. O resto do exemplo é semelhante a uma `BooleanQuery` normal.

```
public void testBoostQuery() {
    BuscadorArquivosLocais buscador =
            new BuscadorArquivosLocais();
    Query queryNuvem = new TermQuery(
            new Term("conteudo", "nuvem"));
    Query queryRede = new TermQuery(
            new Term("conteudo", "rede"));
    BoostQuery boostQuery = new BoostQuery(queryRede,
            2);
    Query query = new BooleanQuery.Builder()
            .add(queryNuvem, Occur.SHOULD)
            .add(boostQuery, Occur.SHOULD).build();
    buscador.buscar(query);
}
```

RegexQuery

As consultas com regex utilizam a classe `RegexQuery` e a sintaxe não sofre muitas modificações. Utilize qualquer uma das expressões regulares vistas no capítulo com a sintaxe a seguir, onde recuperamos os documentos com o padrão `.*bug.{1,5}[0-9]{4}@`. Aqui temos de retirar as barras (`/`), que devem ser usadas apenas na consulta com sintaxe clássica.

```
// Formato de um item do tipo Bug
String regex = ".*bug.{1,5}[0-9]{4}@";
Term termo = new Term("conteudoNaoAnalisado", regex);
```

FuzzyQuery

A `FuzzyQuery` é uma busca por um termo aproximado, com um certo nível de imprecisão, por exemplo, quando a grafia da palavra não está exata e tem alguns caracteres incorretos. No exemplo a seguir vamos pesquisar por documentos que contêm uma palavra similar a `manuel`, incluindo `manoel`, `manual` etc. Observe que essa consulta é equivalente a usar o `QueryParser` com a sintaxe `conteudo:manuel~`. O

padrão para a quantidade de edições é 2. Veja no exemplo como fica com a API.

```
Term termo = new Term("conteudo", "manuel");
FuzzyQuery query = new FuzzyQuery(termo);
```

O exemplo visto utiliza até 2 transformações, ou seja, 2 mudanças de letras para encontrar os documentos correspondentes. A `FuzzyQuery` tem um método construtor que indica a quantidade de edições, que pode ser 0, 1 e 2. Vamos testar com 1 transformação. Entenda que a quantidade de itens tem de ser menor que com 2 transformações.

```
Term termo = new Term("conteudo", "manuel");
FuzzyQuery query = new FuzzyQuery(termo, 1);
```

As transformações de caracteres podem ocorrer em qualquer parte da palavra, o que é muito abrangente. O próximo construtor da `FuzzyQuery` indica a quantidade de caracteres no início do termo que devem ser preservados antes de começar as transformações. No exemplo a seguir, o método construtor define que o Lucene só vai começar a transformar a palavra depois dos 5 primeiros caracteres, ficando assim uma consulta ainda mais restrita.

```
Term termo = new Term("conteudo", "manuel");
FuzzyQuery query = new FuzzyQuery(termo, 1, 5);
```

Consultando campos numéricos

Há um novo tipo de dado para indexar campos numéricos de forma eficiente. O `LongPoint` é uma novidade no Lucene e a forma de consultá-lo também é diferente. Neste caso, temos de usar a API, como no exemplo a seguir. O campo indexado como `LongPoint` é o `tamanhoLong` , não confundir com o campo `tamanho` , que foi indexado como String, mesmo que seu conteúdo seja numérico. Esta consulta não tem um correspondente na sintaxe clássica porque o `LongPoint` é um tipo novo de dado.

O bloco de código a seguir encontra os documentos com tamanho entre 0 e 500.000 bytes:

```
Query query = LongPoint.newRangeQuery("tamanhoLong",
        0, 500000);
```

Além do `LongPoint` para campos do tipo *long* temos outros tipos:

- `BinaryPoint` : para dados binários (bytes);
- `DoublePoint` : para *double*;
- `IntPoint` : para inteiros;
- `FloatPoint` : para ponto flutuante.

Resumo

Neste capítulo, conhecemos diversos tipos de busca, iniciando com as tradicionais consultas por palavra-chave até chegarmos em combinações complexas como `FuzzyQuery` e expressões regulares. Com esse conhecimento você já pode usar a grande maioria das buscas.

A sintaxe clássica e a API têm o mesmo propósito, que é a definição dos critérios de busca. Com as duas alternativas podemos recuperar o mesmo conjunto de dados. Contudo, em alguns casos específicos apenas a API resolve, como na consulta com `LongPoint` e `SpanNearQuery` .

O próximo capítulo fala sobre as principais classes do Lucene e veremos opções avançadas dos recursos utilizados até agora. Entraremos nos detalhes de classes como `Analyzer` , `Directory` e `IndexWriter` para conferir as outras possibilidades de indexação e análise de texto.

O objetivo é conhecer profundamente a configuração do índice, como fazer melhorias na performance da aplicação e otimização de uso do disco, como é o funcionamento interno do analisador de texto, inclusive em português, como é feito o cálculo de similaridade e, por fim, opções avançadas de ordenação.

Capítulo 5
Principais classes do Lucene

Neste capítulo vamos analisar com mais detalhes as principais classes do Lucene, incluindo os métodos, os parâmetros e as configurações. No decorrer do capítulo, as classes aparecem seguindo uma ordem de importância dentro de um projeto Lucene, assim, os primeiros tópicos são essenciais para toda aplicação. Os itens no final do capítulo oferecem ajustes finos que só fazem diferença em sistemas de busca complexos ou se você quer aprender como é o funcionamento interno do Lucene.

Alguns dos recursos estudados neste capítulo são experimentais. Significa que eles podem sofrer alterações em cada versão do Lucene. As mudanças podem ser apenas para facilitar a criação de objetos, como um novo método construtor ou a inclusão de uma classe `Builder`. Entretanto, pode ser algo um pouco mais complicado, como a inclusão de novas classes e, o pior de todos os pesadelos do programador, que é a exclusão de classes, obrigando a equipe a fazer a refatoração do código.

Em muitos casos, as classes/interfaces têm diversas subclasses, criando uma hierarquia. Por exemplo, a classe `Directory`, que representa uma lista de arquivos e diretórios, tem várias subclasses, mas vamos citar apenas as duas mais usadas, que são o `MMapDirectory` e o `NIOFSDirectory`. Tudo isso será estudado, começando pela classe `Document`.

5.1 Document

O `Document` é a classe básica para indexação e busca no Lucene, armazenando um conjunto de campos e valores textuais. Todos os

itens indexados se transformam em documentos, incluindo registros do banco de dados, arquivos de texto, comentários e páginas web. Ou seja, o Lucene indexa documentos. É o `Document` que faz a tradução entre a sua informação e o índice do Lucene. A informação está sempre em formato texto, enquanto o índice está em um formato proprietário do próprio Lucene. Assim, o documento é o recurso com o qual você configura o que será indexado e como será armazenado no índice.

A classe `Document` apresenta métodos para adicionar, remover e recuperar os valores que serão indexados. Perceba que é uma classe *final*, isto é, não podem existir subclasses de `Document`. Essa estratégia é usada para evitar comportamentos diferentes daqueles definidos originalmente na classe. De outra forma, o programador (você) poderia reescrever um dos métodos, ocasionando comportamento inesperado no sistema. Por exemplo, se não fosse *final* você poderia mudar a forma de adicionar valores no documento, e ninguém sabe como seria o resultado disso se fosse escrito um código ruim.

Cada documento contém uma lista de campos, sendo que um desses campos pode identificar unicamente o documento. É algo como um ID ou chave-primária. Por exemplo, se estiver indexando um banco de dados, para cada campo da tabela teríamos um campo no documento. No caso de outros tipos de fonte de dados, como um arquivo binário (Word, PDF, Excel), não nos referimos apenas ao seu texto, mas também aos seus metadados. Metadados são dados informativos sobre um arquivo, como data de criação, título, autor, assunto, palavras-chave, tamanho em bytes e quantidade de páginas. Existem mais algumas técnicas que são discutidas no capítulo *9. Recursos avançados*.

Há várias formas de se criar uma instância de documento e a sintaxe mais simples seria a que está logo a seguir, que usa `TextField` como atalho para adicionar os campos no documento. Outros métodos para criar documentos serão apresentados mais

tarde no capítulo, depois de serem apresentadas as demais classes do Lucene.

```
Document doc = new Document();
TextField campoNome = new TextField("nome",
        "marco antonio", Store.YES);
doc.add(campoNome);
TextField campoEndereco = new TextField("endereco",
        "rua 37 sul", Store.YES);
doc.add(campoEndereco);
```

Nesse bloco, temos o exemplo de como criar e adicionar campos em um documento. No caso, foram adicionar os campos *nome* e *endereco*, com os respectivos valores *marco antonio* e *rua 37 sul*. Na lista a seguir podemos ver mais alguns dos métodos:

Método	Descrição
`clear()`	Remove os campos do documento.
`get(nome-do-campo)`	Recupera o valor do campo em formato String. No caso de campos numéricos faz a conversão para String.
`getBinaryValue(nome-do-campo)`	Recupera um *array* de *bytes* com o valor do campo indicado.
`getBinaryValues(nome-do-campo)`	Recupera um *array* de *bytes* com os valores dos campos indicados.
`getField(nome-do-campo)`	Recupera o campo do documento. Se houver vários campos com o mesmo nome, retorna o primeiro.
`getFields()`	Recupera uma lista com os campos do documento.
`getValues(campo)`	Retorna um *array* com os valores do campo multivalorado.

Método	Descrição
`iterator()`	Returna um *iterator* com os campos do documento.
`removeField(nome-do-campo)`	Remove o campo desta instância do documento. *Não* remove o campo do índice, já que o Lucene não permite alteração.

5.2 Field

Um documento do Lucene é uma lista de campos, ou seja, um `Document` é representado por uma coleção de objetos `Field` agrupados, sendo que cada campo tem nome, tipo e valor. O tipo do campo pode ser textual, numérico ou binário. Na grande maioria dos casos, um sistema de busca está manipulando conteúdo textual. Nestes casos, nós usamos a classe `TextField`, que é adequada para texto em geral.

O `TextField` está para o Lucene assim como o String está para a linguagem Java, assim, pode ser usado de forma genérica. No `TextField`, o analisador do Lucene vai transformar o texto em letras minúsculas, eliminar as *stop words* e remover caracteres especiais, (como ponto, vírgula) e símbolos (como percentual). Para situações diferentes o Lucene oferece classes especializadas, como visto na listagem a seguir:

Classe	Descrição

Classe	Descrição
`TextField`	Tipo mais comum de campo do Lucene. Atende bem à maioria das situações envolvendo texto em geral.
`StringField`	Indexa valores que não podem ser separados e são usados para identificação, como CPF, telefone e data.
`BinaryPoint`, `DoublePoint`, `FloatPoint`, `IntPoint` e `LongPoint`	Permitem a indexação, respectivamente, de valores binários (*bytes*), `double`, `float`, `int` e `long`. São otimizados para busca deste tipo de dado.
`LatLonPoint`	Indexa valores de latitude e longitude.
`StoredField`	Representa um campo que será indexado e seu conteúdo será armazenado no índice.
`InetAddressPoint`	Indexa o endereço IP (IPv4 e IPv6) e executa a busca por intervalos.
`SuggestField`	Utilizado no módulo de recomendação, para sugestão de documentos semelhantes.
`FacetField`	Utilizado na busca facetada (tem uma seção sobre o tema no final do livro).
`SortedDocValuesField`	Campos otimizados para tarefas de *scoring*, ordenação, *faceting*, *grouping* e *joining*. Serão estudados no capítulo de *9. Recursos avançados*, assim como seus similares, `SortedNumericDocValuesField` e `NumericDocValuesField`.

Os métodos do `Field` estão detalhados nos próximos itens ao longo desta seção. Inicialmente, vamos apenas listá-los e no decorrer do texto veremos os exemplos práticos. Os métodos são:

Método	Descrição
`fieldType()`	Descreve as propriedades do campo.
`name()`	Nome do campo.
`tokenStream(...)`	Cria uma instância de `TokenStream`, classe que faz a separação das palavras do campo.
`binaryValue()`, `numericValue()`, `readerValue()`, `stringValue()`	Retorna o valor do campo, que pode ser binário, numérico, String ou um `Reader`.

FieldType

A classe `FieldType` representa as propriedades de cada campo no documento indexado. É onde dizemos se o campo será armazenado no índice, analisado pelo `Analyzer`, se deve armazenar o vetor de termos etc. Esses atributos são verificados antes da indexação, para informar ao Lucene como os valores devem ser processados. Voltaremos a esse assunto no capítulo *9. Recursos avançados*.

As opções do `FieldType` alteram a forma de tratar os valores do campo, o que se reflete no tamanho do índice, na velocidade da indexação e na performance da busca. Por exemplo, um campo marcado como numérico precisa de menos espaço de armazenamento, enquanto um campo texto marcado com a opção `DOCS_AND_FREQS_AND_POSITIONS_AND_OFFSETS` precisa de mais espaço no disco. A seguir estão as opções disponíveis do `FieldType`:

Método	Descrição

Método	Descrição
`docValueType()`	Especifica se o campo é numérico, binário ou ordenado.
`indexOptions()`	Define a quantidade de informações que será armazenada no índice, por meio do `IndexOptions`. A lista dessas opções está logo adiante.
`omitNorms()`	Indica se o campo omite a normalização do texto, um valor usado para equilibrar os documentos com quantidade de termos muito diferentes.
`pointDimensionCount()` e `pointNumBytes()`	Quando tem valor maior que zero indica que o campo é armazenado como um ponto com várias dimensões.
`stored()`	O conteúdo do campo é armazenado no índice.
`storeTermVectors()`	Indica se o campo armazena os termos de um campo em um vetor, criando um índice invertido dentro do próprio campo. É útil quando o campo contém texto muito longo.
`storeTermVectorOffsets()`	Indica se o campo armazena a posição inicial e final dos caracteres de cada termo (*offset*).
`storeTermVectorPositions()`	Indica se o campo armazena a posição (sequência) de cada termo dentro do texto.

Método	Descrição
`storeTermVectorPayloads()`	Indica se o campo armazena o peso de cada termo dentro do texto. Pode ser usado na consulta, para aumentar a pontuação de um documento que contém palavras-chave importantes.
`tokenized()`	Indica se o campo deve ser processado pelo `Analyzer`.

E aqui está a lista de opções do `IndexOptions`. De uma forma geral, ele define o nível de informação que será gravada sobre um determinado campo:

Atributo	Descrição
`NONE`	O campo não será indexado.
`DOCS`	Apenas os termos (as palavras do documento) são indexados.
`DOCS_AND_FREQS`	Os documentos e a frequência de cada um dos termos é indexado.
`DOCS_AND_FREQS_AND_POSITIONS`	Indexa documentos, frequências e posições dos termos. É o valor padrão.

Atributo	Descrição
`DOCS_AND_FREQS_AND_POSITIONS_AND_OFFSETS`	Indexa todas as informações do texto, incluindo os termos, as frequências, as posições e os deslocamentos dos caracteres dos termos (*offsets*).

O `FieldType` é usado para personalizar as opções de um campo. Por exemplo, vamos considerar o `TextField`, que contém um `FieldType` para definir quais as características deste tipo de campo. Assim, o `FieldType` associado ao `TextField` tem estas características:

Atributo	Descrição
`stored`	O valor textual do campo é gravado no índice, o que permite sua posterior recuperação.
`indexed`	Indica que esse campo será indexado e, portanto, será pesquisável.
`tokenized`	O conteúdo será processado pelo `Analyzer`.

Mas podemos usar diretamente o `FieldType`, simulando o comportamento de um `TextField`. Veja o código a seguir:

```
FieldType ftText = new FieldType();
ftText.setIndexOptions(
        IndexOptions.DOCS_AND_FREQS_AND_POSITIONS);
ftText.setStored(true);
```

Outra classe bastante usada é a `StringField`. Neste caso, a busca deve ser feita pelo valor exato do campo, como é o caso de uma busca por CPF, ID, data, UF etc. As características do `StringField` são as seguintes:

Atributo	Descrição
`stored`	O conteúdo também é armazenado no índice.
`indexed`	Indica que este é um campo pesquisável.
`omitNorms`	Não considera a normalização do texto. Como a busca é exata, não precisa fazer os cálculos para relevância.
`indexOptions=DOCS`	Apenas os termos são indexados. Novamente, como a busca é exata não precisa guardar detalhes do campo.

Vamos ver um exemplo de como usar cada uma delas, a `TextField` e a `StringField`. O código a seguir cria um documento com dois campos, o primeiro é um telefone e o segundo é um endereço. Um telefone é consultado pelo seu valor exato, ou seja, é um `StringField`. Por outro lado, um endereço costuma ser pesquisado por uma parte de sua descrição, assim, é um `TextField`. Veja como fica:

```
public void criarDocumentoTextField() {
    Document doc = new Document();
    TextField campoEndereco = new TextField("endereco",
            "rua das flores número 13 ", Store.YES);
    doc.add(campoEndereco);
    System.out.println("Tipo: " + campoEndereco.fieldType());
    StringField campoTelefone = new StringField("telefone",
            "81194625", Store.YES);
    doc.add(campoTelefone);
    System.out.println("Tipo: " + campoTelefone.fieldType());
}
```

Nesse bloco, o `ftText` tem a mesma função de um `TextField`. De forma semelhante, um `StringField` é configurado como sendo `stored`, `indexed`, `omitNorms` e `indexOptions=DOCS` e também armazena

essas configurações em um `FieldType`. Veja agora como fica a simulação de um `StringField` com um `FieldType`:

```
FieldType ftString = new FieldType();
ftString.setIndexOptions(IndexOptions.DOCS);
ftString.setStored(true);
ftString.setTokenized(false);
ftString.setOmitNorms(true);
```

Assim, temos o `ftString`, que simula o `StringField`, e o `ftText`, que simula o `TextField`. Depois de definidos, podemos usar esses tipos nos campos de um documento para indexação, como no bloco de código a seguir:

```
Document doc = new Document();
Field campoEndereco = new Field("endereco",
        "rua das flores número 13", ftText);
doc.add(campoEndereco);
Field campoTelefone =
        new Field("telefone", "81194626", ftString);
doc.add(campoTelefone);
```

Dessa forma, vimos que é possível mudar a configuração de cada campo do índice, sem nos limitar aos campos disponíveis no Lucene. Claro que neste caso estamos escrevendo mais código, mas há situações em que as classes existentes não atendem aos requisitos, como será abordado no capítulo *9. Recursos avançados*.

5.3 IndexWriter

O `IndexWriter` é usado para adicionar e excluir documentos do índice. Quando um documento é adicionado por meio do `addDocument`, ele fica na RAM até que o limite do *buffer* seja atingido. O padrão utilizado no Lucene é 16 MB. Com isso, quando há mais de 16 MB de documentos armazenados na memória, é criada uma

estrutura temporária chamada `Segment`, onde ficam armazenados os dados enquanto não são gravados no disco.

Um documento estará disponível para a consulta após uma chamada ao método `IndexWriter.commit()` ou `IndexWriter.close()`. A exclusão usa o mesmo princípio: o documento será removido após chamar o `commit` ou o `close`. Neste momento, o `IndexWriter` faz a gravação do índice no disco, uma operação muito custosa em termos de performance.

> Isso é parcialmente verdade, porque o Lucene tem consultas em tempo quase real, ou NRT (*Near Real-Time*), sem a necessidade do `commit`. Este recurso especial será visto mais tarde neste capítulo.

Os principais métodos do `IndexWriter` são:

Método	Descrição
`addDocument`	Adiciona o documento no índice.
`deleteDocument`	Remove documentos do índice através de uma consulta ou termo.
`updateDocument`	Atualiza o documento, removendo e adicionando-o no índice.
`commit`	Grava as alterações no índice, tanto as inclusões como as exclusões de documentos.
`close`	Grava as alterações no índice e fecha os recursos utilizados.

Algumas vezes precisamos mudar a configuração do indexador para otimizar a indexação ou a busca. Isso é feito em função do equipamento e dependente da velocidade e quantidade de memória e do disco disponíveis. No geral, não temos recursos infinitos de

hardware, o que nos obriga a fazer algum tipo de otimização quando a demanda pela aplicação aumenta.

> Observação 1: apesar de existir o método `updateDocument`, não existe efetivamente a atualização de um Document. Mesmo usando `updateDocument`, o que acontece é a exclusão do documento antigo e a inclusão de um item com os novos valores.
>
> Observação 2: o Lucene é otimizado por padrão e, a cada nova versão, essas otimizações são revistas e melhoradas. Evite mudar as configurações a menos que saiba exatamente o que está fazendo.

Teste de bloqueio do IndexWriter

A gravação do índice só pode ser feita por um `IndexWriter` de cada vez, ou seja, não podem ser feitas duas gravações ao mesmo tempo. Este é o mecanismo do Lucene para evitar a corrupção do índice. Funciona assim: após aberto, o `IndexWriter` cria um arquivo chamado `write.lock` no diretório que está sendo usado para indicar que já está sendo realizada uma gravação.

Dessa forma, somente uma instância da classe pode ser usada para gravar neste diretório. Se tentar criar mais um `IndexWriter` apontando para o mesmo índice será lançada a exceção `LockObtainFailedException`, indicando que o diretório não pôde ser bloqueado para a gravação do índice.

A classe `WriteLockTest` foi criada para testar essa situação, onde temos duas instâncias de `IndexWriter` tentando gravar no mesmo diretório. Foram criados os objetos `writer` e o `segundoWriter`, ambos tentando gravar no mesmo diretório. Quando o `segundoWriter` tenta abrir o diretório para gravação é lançada a `LockObtainFailedException`, como era esperado.

No final da classe, verificamos o estado do `writer`, que deve estar aberto, e o `segundoWriter`, que deve estar nulo porque não pode haver dois `IndexWriter` apontando para o mesmo diretório.

```
public class WriteLockTest {
    @Test
    public void testWriterDuplo() throws IOException {
        IndexWriter writer = null;
        IndexWriter segundoWriter = null;
        String caminho =
                System.getProperty("java.io.tmpdir") + "/temp";
        Directory diretorio =
                FSDirectory.open(Paths.get(caminho));
        Analyzer analyzer = new StandardAnalyzer();
        IndexWriterConfig conf = new IndexWriterConfig(analyzer);
        IndexWriterConfig segundaConf =
                new IndexWriterConfig(analyzer);
        // Tenta abrir o diretório para gravação
        try {
            writer = new IndexWriter(diretorio, conf);
            segundoWriter =
                    new IndexWriter(diretorio, segundaConf);
        } catch (LockObtainFailedException e) {
            e.printStackTrace();
        }
        // Verifica o estado dos writers
        assertNotNull(writer);
        assertNull(segundoWriter);
    }
}
```

Após a execução do teste, é possível verificar uma mensagem de erro disparada pelo `e.printStackTrace()` para confirmar que não pode haver dois `IndexWriter` no mesmo diretório. Esta é a mensagem:

```
org.apache.lucene.store.LockObtainFailedException: Lock held by this
virtual machine: /tmp/temp/write.lock
```

Isso nos leva à conclusão de que realmente não é possível criar dois objetos `IndexWriter` apontando para o mesmo diretório, que era

o objetivo da classe `WriteLockTest`.

IndexWriterConfig

As configurações do `IndexWriter` são feitas por meio de um `IndexWriterConfig`. Aqui, os parâmetros obrigatórios são o `Analyzer`, o `Directory` e o `IndexWriterConfig`. Esses itens aparecem em sequência no código a seguir e representam o conjunto mínimo de parâmetros necessários para criar um `IndexWriter`. Veja a sintaxe:

```
    Analyzer analyzer = new StandardAnalyzer();
    diretorio = FSDirectory
            .open(Paths.get(diretorioIndice));
    IndexWriterConfig conf = new IndexWriterConfig(analyzer);
    writer = new IndexWriter(diretorio, conf);
```

Uma vez que aplicamos esses parâmetros, obtemos o objeto `writer`, como visto. Existem outros parâmetros que serão vistos mais para a frente, mas por enquanto veremos a configuração do `writer` atual. Veja os parâmetros e seus valores:

```
analyzer=org.apache.lucene.analysis.standard.StandardAnalyzer
ramBufferSizeMB=16.0
maxBufferedDocs=-1
maxBufferedDeleteTerms=-1
mergedSegmentWarmer=null
openMode=CREATE_OR_APPEND
similarity=org.apache.lucene.search.similarities.BM25Similarity
codec=Lucene60
mergePolicy=[TieredMergePolicy: maxMergeAtOnce=10,
maxMergeAtOnceExplicit=30, maxMergedSegmentMB=5120.0, floorSegmentMB=2.0,
forceMergeDeletesPctAllowed=10.0, segmentsPerTier=10.0
{Continua...}
```

Alguns atributos do `IndexWriterConfig` merecem atenção e fazem diferença para a performance da indexação, como veremos nas próximas seções deste capítulo:

Método	Descrição

Método	Descrição
`openMode.CREATE`	Cria ou sobrescreve um índice.
`openMode.APPEND`	Abre um índice preexistente.
`openMode.CREATE_OR_APPEND`	Cria ou abre o índice.
`setMaxBufferedDocs`	Quantidade de documentos em memória antes de criar um novo segmento.
`setRAMBufferSizeMB`	Representa a quantidade de memória RAM usada para armazenar documentos antes de serem gravados no índice.
`setSimilarity`	Calcula a ordenação do resultado das buscas. Por padrão, é o `BM25Similarity`.
`setUseCompoundFile`	Grava os novos segmentos em um arquivo separado, um *compound file*.

Os parâmetros `maxBufferedDocs` e `ramBufferSizeMB` definem a estratégia usada para criar novos segmentos. Com eles, você escolhe se os segmentos do índice serão gravados com base no número de documentos no *buffer* (`maxBufferedDocs`) ou pela quantidade de memória utilizada (`ramBufferSizeMB`). Você só pode escolher uma das opções.

A última configuração para a criação do índice com o `IndexWriterConfig` que veremos é o `mergePolicy` , que é a política usada para juntar os arquivos dos segmentos. O padrão é o `TieredMergePolicy` , que tenta criar os arquivos aproximadamente com o mesmo tamanho. Outros exemplos serão vistos durante o capítulo.

Vale lembrar que fazer o teste de carga é sempre uma boa ideia para evitar surpresas quando a aplicação entrar em produção, principalmente no caso de projetos web, onde a quantidade de acessos pode aumentar rapidamente. Assim, antes de entrar em produção, é importante testar a aplicação com a quantidade de documentos que se espera em uma situação real.

5.4 Directory

`Directory` é uma classe abstrata que representa, como o nome sugere, um diretório e sua lista de arquivos. O Lucene usa a política de *write once* e, uma vez que você faz o `commit`, os documentos em memória são gravados em um segmento, que é um arquivo inalterável.

Um segmento só pode ser aberto para leitura ou exclusão. A leitura no segmento é aleatória, ou seja, o Lucene pode recuperar os documentos em qualquer parte do arquivo e essa é uma operação bastante rápida.

A classe abstrata `FSDirectory` é a base para acessar o diretório e a forma mais comum de utilização é essa:

```
String caminho = "...";
Directory diretorio = FSDirectory
        .open(Paths.get(caminho));
```

O método `open` vai procurar a melhor implementação para o sistema operacional do servidor. Atualmente, retorna um `MMapDirectory` para plataformas de 64 bits como Solaris, Linux e Windows. Para Windows 32 bits (sim, ele ainda existe) retorna o `SimpleFSDirectory`, implementação com baixa performance. Nos demais casos em que o sistema operacional não suporta o `MMapDirectory`, retorna um `NIOFSDirectory`.

Se tiver um bom motivo para isso, você pode escolher outra implementação do `Directory` instanciando diretamente a classe que deseja. O bloco de código a seguir mostra como criar um diretório com o `NIOFSDirectory` .

```
NIOFSDirectory diretorio = new NIOFSDirectory(
        Paths.get(caminho));
```

5.5 IndexReader

O `IndexReader` é responsável pelo acesso aos documentos disponíveis no índice em determinado instante. Quando aberto, um `IndexReader` recupera a lista de segmentos disponíveis naquele momento. A sintaxe a seguir mostra como abrir um `IndexReader` :

```
IndexReader reader = DirectoryReader
        .open(diretorio);
```

Agora, veja na figura o papel de cada elemento dentro do processo de busca. O `IndexReader` é um recurso intermediário que fica entre o `Directory` e o `IndexSearcher` , que executa a consulta do usuário.

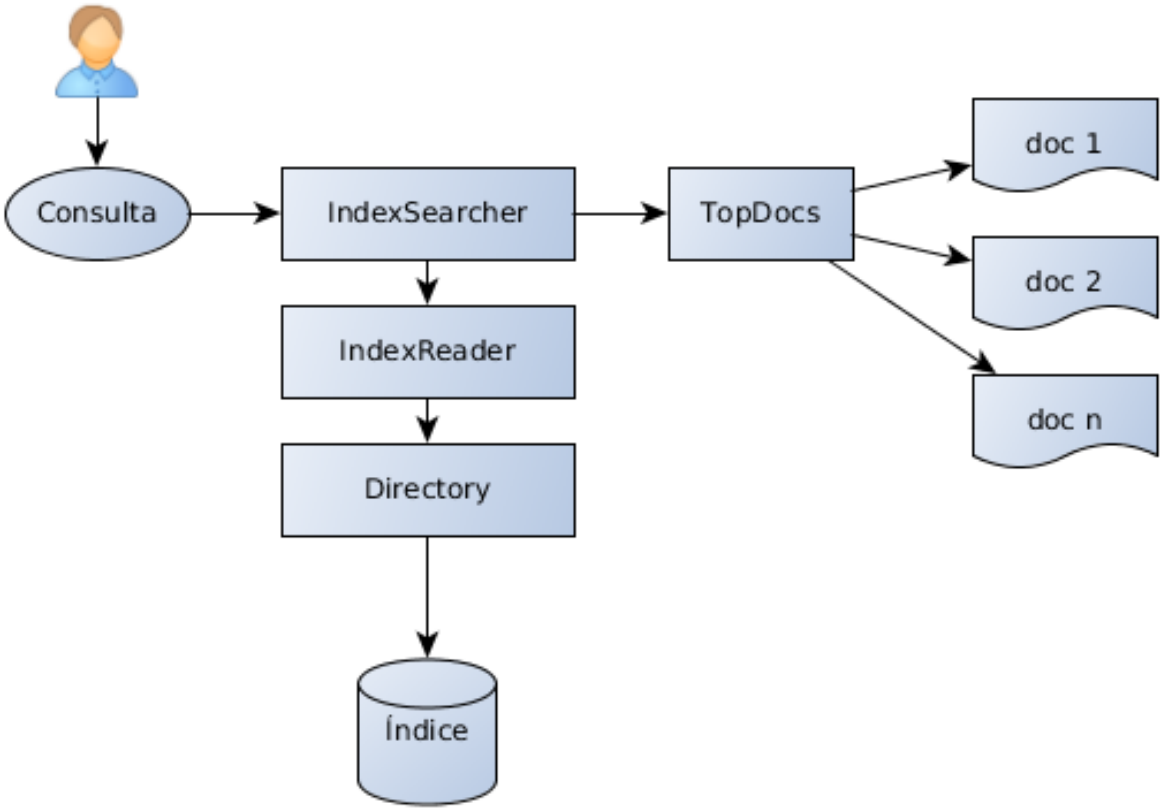


Figura 5.1: Estrutura do Index Reader/Searcher.

Até agora tivemos um índice estático, sem alterações, o que não é verdade nos sistemas em produção. Vamos mudar um pouco essa abordagem fazendo consultas e inclusões de documentos ao mesmo tempo.

Para tanto, é natural que a consulta e a indexação posam ser executadas simultaneamente, quer dizer, o usuário pode realizar buscas e o indexador pode adicionar e excluir documentos conjuntamente.

O que pode variar é a velocidade com que as alterações ficam disponíveis. Em um sistema que usa SGBDR, por exemplo, quando um registro é inserido, imediatamente ele está disponível para o usuário.

Buscas em tempo quase real

Em soluções de *big data*, como o nosso motor de buscas, é comum utilizar o conceito de aplicação em tempo *quase* real, ou *near real-time*, ou apenas NRT. Os itens inseridos ou excluídos estarão disponíveis *quase* imediatamente. Essa diferença de tempo pode ser de milésimos de segundo até algumas horas.

É o que acontece quando o `IndexReader` fica aberto muito tempo sendo compartilhado por múltiplos usuários, enquanto o `IndexWriter` está adicionando e excluindo novos itens no índice. Para ter acesso a essas atualizações, teríamos de reabrir o índice, sendo que esta é uma operação demorada.

Abrir o índice é uma operação "cara" porque acesso o disco, que tem acesso lento. Nas versões mais recentes, o Lucene implementou uma alternativa melhor, que é usar o método `DirectoryReader.open(IndexWriter)` para realizar buscas NRT, que acessa a memória, um meio de acesso rápido. Veja que esse método permite a abertura do índice usando o `IndexWriter` que estava processando novos itens.

Essa é a busca NRT, ou busca em tempo quase real. Assim, conseguimos recuperar os documentos mesmo sem executar o método `commit`, porque o `IndexWriter` sabe quais documentos incluiu ou excluiu. Nunca vai ser em tempo real porque há uma latência que o Lucene não pode controlar quando os segmentos estão sendo sincronizados com o sistema operacional, principalmente em segmentos grandes.

Comparando com a situação anterior, na qual não conhecíamos o recurso NRT, para ter acesso às atualizações teríamos que chamar novamente o `DirectoryReader.open(Directory)`. Perceba que o parâmetro usado é um `Directory`, que tem de ler todos os segmentos do índice.

O código a seguir mostra como funciona uma busca em tempo quase real. É o que faz a classe `TesteNRT`. Ela consulta a quantidade de itens do índice antes e depois de adicionar novos documentos. Note que o `writer` não chamou o método `commit`, mesmo assim os novos itens estão disponíveis para consulta através da instância de `novoReader`.

```
public class TesteNRT {
    private static String DIRETORIO_INDICE = System
            .getProperty("user.home") + "/livro-lucene/indice";

    @Test
    public void testeTempoReal() throws IOException {
        // Cria os objetos normalmente
        Directory diretorio = FSDirectory
                .open(Paths.get(DIRETORIO_INDICE));
        IndexWriter writer = new IndexWriter(diretorio,
                new IndexWriterConfig(new StandardAnalyzer()));
        IndexReader readerAnterior = DirectoryReader
                .open(diretorio);
        IndexSearcher searcher = new IndexSearcher(
                readerAnterior);
        TopDocs hits = searcher.search(new MatchAllDocsQuery(),
                1);
        // Guarda a quantidade de itens
        long numDocsAnterior = hits.totalHits;
        // Adiciona documento e não faz commit
        writer.addDocument(criaDocumento());
        writer.addDocument(criaDocumento());
        // Abre um reader NRT através do writer
        IndexReader novoReader = DirectoryReader
                .open(writer);
        searcher = new IndexSearcher(novoReader);
        hits = searcher.search(new MatchAllDocsQuery(), 1);
        long numDocsAtual = hits.totalHits;
        // Verifica se tem um item a mais
        assertTrue(numDocsAtual == numDocsAnterior + 2);
        novoReader.close();
        readerAnterior.close();
        writer.close();
```

```
        diretorio.close();
    }

    private Iterable<? extends IndexableField> criaDocumento() {
        Document doc = new Document();
        doc.add(new TextField("conteudo", "Teste em tempo real",
                Store.YES));
        return doc;
    }
}
```

A grande diferença está na instrução `IndexReader novoReader = DirectoryReader.open(writer)`, que abre o índice a partir do `writer` e não de um `Directory`. Com isso, não precisamos de um `commit` para ver as alterações no índice.

Reabrindo um índice alterado

Além da busca NRT o Lucene tem a opção de reabrir um índice modificado por meio do método `DirectoryReader.openIfChanged(DirectoryReader)`, que abre um novo `IndexReader` apenas se houve mudança no índice desde a última vez em que foi aberto. Se o índice não tiver sido alterado, retorna nulo.

De toda forma, o `IndexReader` original continua aberto e você escolhe qual instância pretende usar. Esse método tende a ser mais rápido do que criar uma nova instância do `IndexReader`. E não se esqueça de fechá-los. Lembre-se que recursos abertos são potencialmente perigosos para qualquer software que roda no *back end*.

A classe `TesteIndiceAlterado` do código adiante mostra como isso é feito. A instrução `DirectoryReader readerAnterior = DirectoryReader.open(writer)` abre um `reader` NRT. Na sequência, a instrução `IndexReader novoReader = DirectoryReader.openIfChanged(readerAnterior, writer)` chama o método `openIfChanged`. Como não houve alteração no índice até esse momento, a primeira verificação `assertTrue(novoReader == null)` garante que o `novoReader` é nulo. Quando um novo documento é

adicionado, a instrução `DirectoryReader.openIfChanged` retorna um `reader` para a versão do índice com a atualização.

Este método é menos dispendioso que o `DirectoryReader.open(diretorio)` porque compartilha recursos já utilizados. Então, quando possível, use `DirectoryReader.openIfChanged` .

```
@Test
public void testeIndiceAlterado() throws IOException {
    // Cria um reader normalmente
    Directory diretorio = FSDirectory
            .open(Paths.get(DIRETORIO_INDICE));
    IndexWriter writer = new IndexWriter(diretorio,
            new IndexWriterConfig(new StandardAnalyzer()));
    DirectoryReader readerAnterior = DirectoryReader
            .open(writer);
    // Guarda a quantidade de itens
    int numDocsAnterior = readerAnterior.numDocs();
    // Abre um reader a partir do anterior
    IndexReader novoReader = DirectoryReader
            .openIfChanged(readerAnterior, writer);
    // O índice não foi alterado, retorna nulo
    assertTrue(novoReader == null);
    // Adiciona documento e não faz commit
    writer.addDocument(criaDocumento());
    // Índice alterado
    novoReader = DirectoryReader
            .openIfChanged(readerAnterior, writer);
    // A quantidade de documentos é maior
    assertTrue(novoReader.numDocs() > numDocsAnterior);
    novoReader.close();
    readerAnterior.close();
    writer.close();
    diretorio.close();
}
```

O `openIfChanged` tem outras variações de parâmetros que estão listadas a seguir.

Método	Descrição

Método	Descrição
`openIfChanged(DirectoryReader oldReader)`	Forma mais básica de usar o método.
`openIfChanged(DirectoryReader oldReader, IndexWriter writer, boolean applyAllDeletes)`	Variação do método visto neste exemplo que pode não considerar as exclusão de documentos.
`openIfChanged(DirectoryReader oldReader, IndexCommit commit)`	Permite indicar apenas o segmento que foi adicionado pelo `writer`.

RAMDirectory

O índice costuma ser armazenado em disco, seja SSD ou HDD. Porém, para pequenos testes podemos usar uma implementação em memória bastante prática. A classe `IndiceEmMemoria`, como o nome sugere, cria um índice em memória que funciona da mesma forma que um índice no disco. Não é uma solução para ser usada com grandes volumes de dados porque estamos limitados à memória disponível na JVM e não há nenhuma grande otimização.

Os dados ficam residentes em memória e, claro, não são persistentes. Depois de terminar a execução do programa os documentos não existem mais. A figura central é o `RAMDirectory`, como pode ser visto no código a seguir. A classe de exemplo tem 3 partes distintas: declaração das variáveis, a inicialização e a criação dos documentos fictícios em memória.

```
public class IndiceEmMemoria {
    private RAMDirectory ramDirectory;
    private IndexWriter writer;

    // Inicializa objetos
    public IndiceEmMemoria() throws IOException {
        ramDirectory = new RAMDirectory();
        IndexWriterConfig conf =
```

```
                new IndexWriterConfig(new StandardAnalyzer());
        writer = new IndexWriter(ramDirectory, conf);
        criarDocumentos();
        writer.close();
    }

    // Cria documentos fictícios
    private void criarDocumentos() throws IOException {
        Document doc = new Document();
        String conteudo = "geralmente você pode usar as "
                + "opções 'mute' ou 'unmute' para "
                + "gerenciar os drivers ALSA no Linux";
        doc.add(new TextField("conteudo", conteudo, Store.YES));
        getWriter().addDocument(doc);
        doc.clear();
        conteudo = "as versões 0.3 e 0.4 têm vários "
                + "problemas no linux "
                + "devido à reestruturação "
                + "da interface do mixer "
                + "que teve de ser reescrito "
                + "em função de problemas"
                + "identificados anteriormente no linux";
        doc.add(new TextField("conteudo", conteudo, Store.YES));
        getWriter().addDocument(doc);
        doc.clear();
        conteudo = "você precisa carregar o módulo "
                + "para o seu cartão de som ou usar "
                + "o utilitário 'kmod' do linux";
        doc.add(new TextField("conteudo", conteudo, Store.YES));
        getWriter().addDocument(doc);
    }

    public IndexWriter getWriter() {
        return writer;
    }

    public RAMDirectory getRamDirectory() {
        return ramDirectory;
    }
}
```

Para testar o índice em memória, criamos a `TesteIndiceEmMemoria`. Veja que não há diferença entre essa classe e as outras que fazem consultas. As classes necessárias são as mesmas: `Directory`, `IndexReader` etc., cada uma delas desempenha seu papel como visto nos exemplos anteriores. A diferença agora é que esses documentos estão apenas na memória.

```
public class TesteIndiceEmMemoria {
    @Test
    public void testeEmMemoria()
            throws IOException, ParseException {
        Directory diretorio = new IndiceEmMemoria()
                .getRamDirectory();
        IndexReader reader = DirectoryReader.open(diretorio);
        IndexSearcher searcher = new IndexSearcher(reader);
        QueryParser parser = new QueryParser("",
                new StandardAnalyzer());
        String consulta = "conteudo:alsa";
        Query query = parser.parse(consulta);
        TopDocs docs = searcher.search(query, 1);
        assertTrue(docs.totalHits > 0);
        diretorio.close();
        reader.close();
    }
}
```

Resumo

Neste capítulo detalhamos o núcleo do Lucene, considerando as configurações iniciais, ou seja, não foram feitos ajustes e refinamentos. Estudamos classes como `Document`, `Field`, `IndexWriter` e `IndexReader`, que fazem parte de uma típica aplicação Lucene.

Apresentamos os tipos de dados disponíveis e como usá-los para enriquecer a aplicação. Vimos que o Lucene oferece o recurso de NRT, que é uma busca em tempo quase real que combina o índice em disco e as alterações feitas em memória. Mostramos também que o índice pode ser armazenado no disco, que é a opção mais

usada, mas que existe a alternativa do `RAMDirectory`, no qual os dados ficam em memória.

No próximo capítulo veremos as configurações avançadas, incluindo técnicas de processamento de texto, técnicas de otimização do índice e os testes para avaliar as diferentes abordagens.

Capítulo 6
Configurações avançadas

Neste capítulo analisaremos as opções avançadas de configuração do Lucene, tendo em vista a otimização. São realizados diversos testes para mostrar quais as diferenças entre as técnicas, de forma que você pode escolher a mais indicada para sua demanda.

Na segunda metade do capítulo apresentamos técnicas de processamento de texto usadas no Lucene, para estender esse conhecimento à criação de classes personalizadas e adaptá-lo para outros modelos de negócio.

6.1 Configurações da indexação

Nos próximos exemplos vamos simular as diferentes configurações usadas na indexação, bem como seus efeitos. Tenha em mente que a indexação é uma tarefa demorada e demanda muito processamento. Para um conjunto pequeno de dados, como um diretório de documentos pessoais, a indexação não chega a ser um problema, mas quando estamos lidando com bases de dados corporativas ou diretórios muito grandes, a performance da indexação merece atenção.

Por exemplo, se a criação de um índice demanda muitas horas, a informação pode não ser útil para o cliente. Dessa forma, quando o volume de dados aumenta significativamente, é normal termos de rever a implementação da indexação. Para analisar essas configurações vamos alterar alguns parâmetros e verificar o resultado nos testes ao longo das próximas seções. Para começar, vale citar que os valores que mais impactam na criação do índice

incluem a memória, o `ramBufferSizeMB` , o `mergePolicy` e o `useCompoundFile` .

Otimizando a indexação

Durante a otimização, vamos alterar as configurações do `IndexWriter` e avaliar os resultados. Nossa metodologia consiste em indexar um diretório com 90.000 arquivos que totalizam 500 MB de texto puro. Depois, vamos verificar o tamanho do diretório do índice gerado e qual o tempo total de indexação.

O computador utilizado tem 16 GB de RAM, processador Intel Core i7 com 4 núcleos, rodando o Debian 8, JDK 1.8.0.161, Lucene 7.4.0 e Eclipse. É essencial que os recursos sejam exatamente os mesmos para todos os testes que faremos. Não faz sentido testar com computadores, sistemas operacionais ou arquivos diferentes e comparar os resultados. Não necessariamente um computador com o dobro de processamento será 2 vezes mais rápido.

Otimizar não significa apenas diminuir o tempo de processamento. Há outras variantes, como quantidade de arquivos e uso de memória, que influenciam na performance geral da aplicação. A indexação é apenas um dos itens que consideramos. Durante esta seção vamos explorar essas opções e ver o comportamento do índice.

O tempo de indexação é otimizado por natureza e reduzi-lo ainda mais é uma tarefa complicada que inclui refatorar o código. Se ainda assim for necessário, há alternativas com o Apache Hadoop, que faz a indexação utilizando processamento distribuído, que é bastante eficiente em termos de tempo.

Wikipedia como base de teste

A Wikipedia disponibiliza suas páginas em formato XML para download e é uma boa fonte de dados para testar a performance de sistemas. O link para os arquivos em português é

https://dumps.wikimedia.org/ptwiki/latest/ e nos nossos exemplos utilizamos a versão mais recente. O arquivo se chama `ptwiki-latest-pages-articles-multistream.xml.bz2` e tem aproximados 1.6 GB.

Descompactado, ele é um arquivo XML de 6 GB com quase 2 milhões de artigos da Wikipedia em português. Não vamos usar tudo isso nos testes porque conseguimos observar os resultados em uma amostra menor de apenas 90.000 itens.

Observação: se não quiser usar a Wikipedia, realize os testes no diretório de documentos que já estava sendo trabalhado no capítulo *3. Indexação e busca*. O importante é que seja uma quantidade grande de arquivos para perceber as diferenças entre as estratégias de indexação.

O Lucene tem um utilitário para extrair o conteúdo da Wikipedia exatamente para este tipo de teste, que é a classe `ExtractWikipedia` do pacote `Lucene Benchmark` . A dependência do Maven é essa:

```
<dependency>
    <groupId>org.apache.lucene</groupId>
    <artifactId>lucene-benchmark</artifactId>
    <version>${lucene.version}</version>
</dependency>
```

Para extrair os artigos da Wikipedia vamos executar a classe com os seguintes parâmetros:

- `-i` : é o *input* do programa. Informe onde está o arquivo do *dump* da Wikipedia.
- `-o` : é o *output* do programa. Informe um diretório de saída.

No Eclipse, isso pode ser configurado no menu `Run -> Run configurations` . Clique na aba *Arguments* e adicione os parâmetros na caixa de texto *Program arguments*, como na imagem:

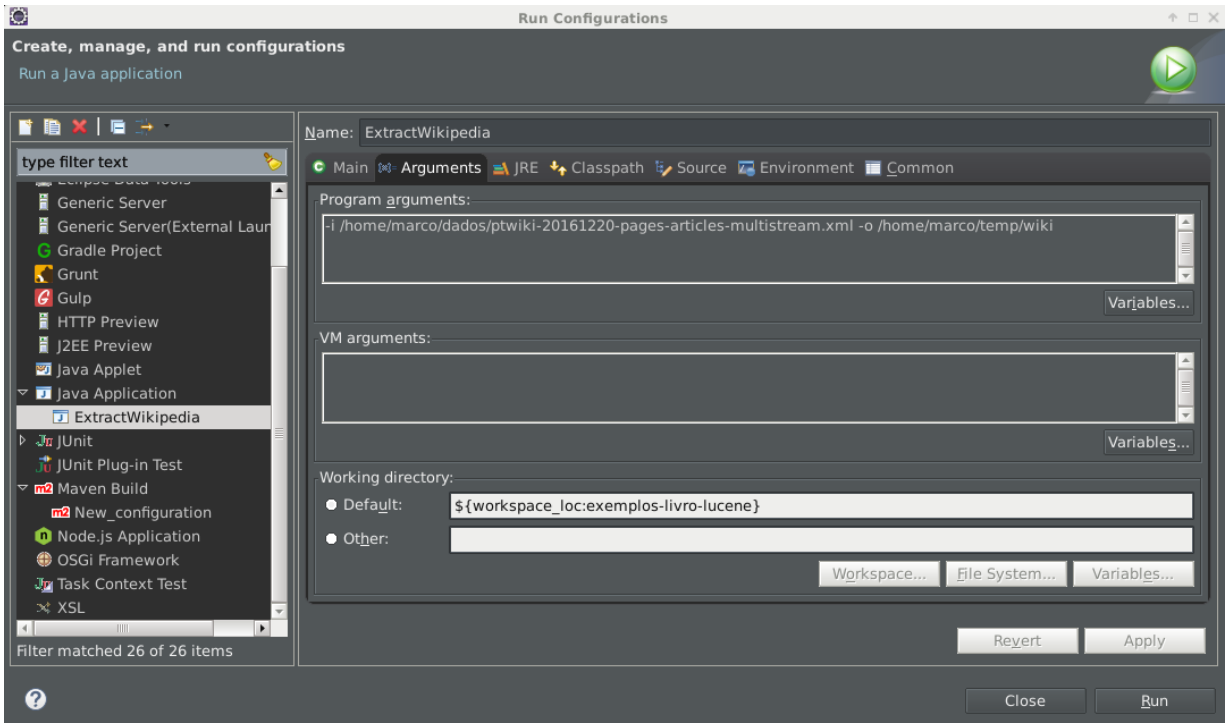
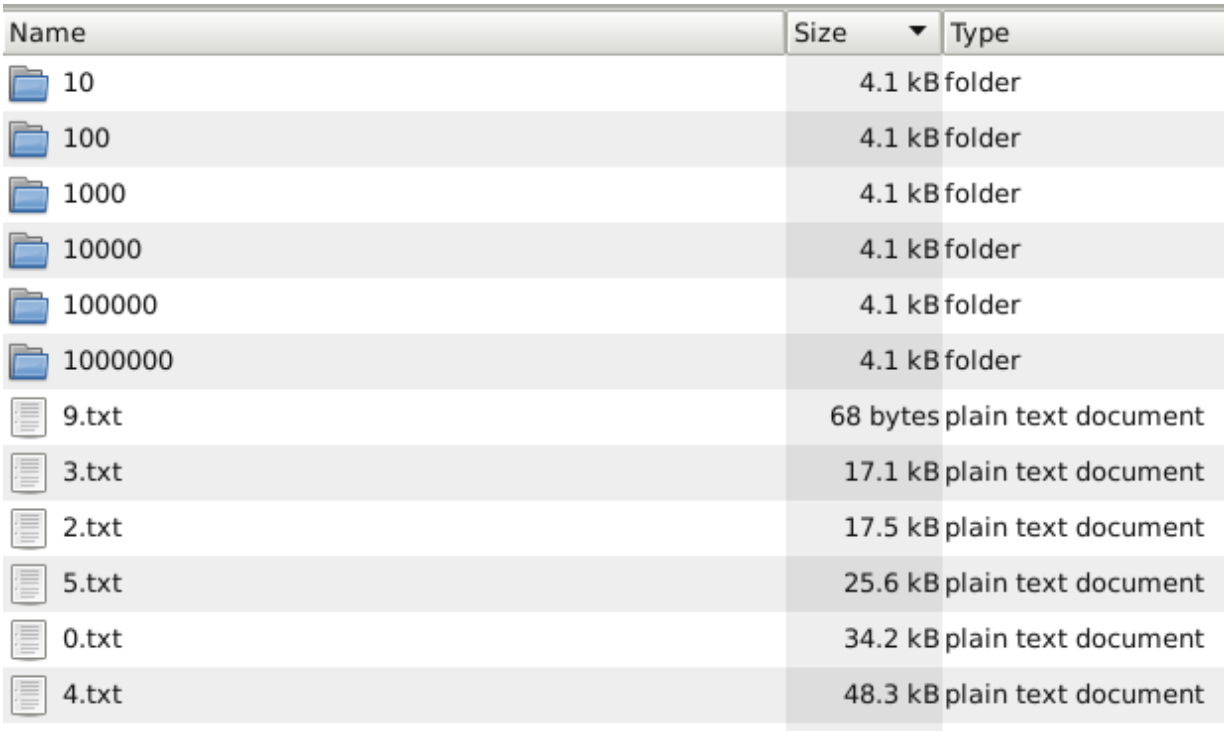


Figura 6.1: Program arguments

Atenção ao diretório temporário de saída, porque ele será excluído. Não aponte esse parâmetro para um diretório com dados reais.

Após a execução do programa será criado um diretório com vários subdiretórios, onde cada artigo da Wikipedia é gravado como um arquivo texto e seu nome é um número sequencial. Veja na imagem a seguir como fica:



Name	Size	Type
10	4.1 kB	folder
100	4.1 kB	folder
1000	4.1 kB	folder
10000	4.1 kB	folder
100000	4.1 kB	folder
1000000	4.1 kB	folder
9.txt	68 bytes	plain text document
3.txt	17.1 kB	plain text document
2.txt	17.5 kB	plain text document
5.txt	25.6 kB	plain text document
0.txt	34.2 kB	plain text document
4.txt	48.3 kB	plain text document

Figura 6.2: Arquivos da Wikipedia

6.2 Performance da configuração padrão

O primeiro passo para avaliar um processo de otimização é coletar as métricas iniciais e depois mudar as configurações para analisar as mudanças na performance do programa.

Começamos com as classes vistas no capítulo *3. Indexação e busca*. A primeira é a classe `IndexadorArquivosLocais`, que não terá alteração. Depois, vamos alterar o diretório de documentos da classe `TesteIndexadorArquivosLocais` e apontá-lo para o diretório dos arquivos extraídos da Wikipedia, ou seja, vamos usar o nosso programa para indexar as páginas da Wikipedia. Lembre-se da imagem anterior que mostra quais são os diretórios extraídos.

Vamos indexar apenas um dos diretórios gerados pelo extrator da Wikipedia, o `10000`, que contém aproximadamente 90.000 arquivos

de texto. É suficiente para o nosso propósito de teste. Se tiver tempo (muitas horas), indexe tudo.

Para todos os testes foi usada a configuração de JVM `-Xmx8g -Xms8g -server` porque o servidor de testes tem 16 GB de RAM. Você precisa adaptar de acordo com o seu hardware, assim, tente usar até metade da memória disponível no seu computador. O resultado da primeira indexação com essa configuração foi este:

- Quantidade de arquivos: 89.996 e deve ser igual em todos os testes;
- Quantidade de bytes indexados: 521 MB e também deve ser igual em todos os testes;
- Tamanho do diretório do índice: 1.1 GB;
- Quantidade de arquivos no índice: 107;
- Tempo de indexação: 175 segundos.

A título de curiosidade eu indexei todos os arquivos da Wikipedia, o que gerou um diretório significativamente grande. O resultado desse processo foi este:

- Quantidade de arquivos: 1.701.421;
- Quantidade de bytes indexados: 4.583 MB;
- Tamanho do diretório do índice: 9.1 GB;
- Quantidade de arquivos no índice: 116;
- Tempo de indexação: 3.600 segundos.

Debug na indexação

A configuração do `IndexWriter` permite o *debug* da indexação de uma forma bem prática por meio do `InfoStream` . Para habilitar esse recurso use a instrução `conf.setInfoStream(System.out)` , que copia o *log* da indexação no console. No bloco de código a seguir, você confere onde fica essa configuração.

```
IndexWriterConfig conf = new IndexWriterConfig(analyzer);
conf.setInfoStream(System.out);
writer = new IndexWriter(diretorio, conf);
```

A primeira parte do *log* está listada a seguir. Perceba que ela mostra a configuração do `IndexWriter`, como o diretório do índice, versão do Lucene e outras coisas que já vimos antes aqui no capítulo.

```
IFD 0 [2016-12-28T01:05:35.689Z; main]: init: current segments file is
"segments";
deletionPolicy=org.apache.lucene.index.KeepOnlyLastCommitDeletionPolicy@11
f0a5a1
IFD 0 [2016-12-28T01:05:35.737Z; main]: delete []
IFD 0 [2016-12-28T01:05:35.738Z; main]: now checkpoint "" [0 segments ;
isCommit = false]
IFD 0 [2016-12-28T01:05:35.738Z; main]: delete []
IFD 0 [2016-12-28T01:05:35.739Z; main]: 1 msec to checkpoint
IW 0 [2016-12-28T01:05:35.740Z; main]: init: create=true
IW 0 [2016-12-28T01:05:35.746Z; main]:
dir=MMapDirectory@/home/marco/livro-lucene/indice-temp
IW 0 [2016-12-28T01:05:35.746Z; main]: MMapDirectory.UNMAP_SUPPORTED=true
{Continua...}
```

Durante a construção do índice, quando a memória atinge o limite configurado, acontece o *flush*, ou seja, os dados são gravados no diretório. Esse processo é mostrado no console para *debug*. Na listagem adiante você pode conferir o momento em que o limite de memória é atingido e quais segmentos são criados.

```
FP 0 [2016-12-28T01:04:38.121Z; main]: trigger flush: activeBytes=16804080
deleteBytes=0 vs limit=16777216
<! --FP 0 [2016-12-28T01:04:38.122Z; main]: thread state has 16804080
bytes; docInRAM=532
FP 0 [2016-12-28T01:04:38.122Z; main]: 1 in-use non-flushing threads
states
DWPT 0 [2016-12-28T01:04:38.201Z; main]: flush postings as segment _0
numDocs=532
IW 0 [2016-12-28T01:04:38.215Z; main]: 13 msec to write norms
IW 0 [2016-12-28T01:04:38.215Z; main]: 0 msec to write docValues
{Continua...}
```

Vale lembrar que o *debug*, bem como o *log* em nível de INFO devem ser usados apenas durante a fase de desenvolvimento, pois todo processo de *debug* gera perda de performance. Não deixe ativo na

produção, a menos que tenha certeza do que está fazendo, por exemplo, para encontrar erros em aplicações distribuídas.

6.3 Tipos de arquivo

Analisando os arquivos gerados pela indexação, encontramos várias extensões diferentes. Veja na imagem alguns desses formatos.

Figura 6.3: Tipos de arquivo

- `segments_X` : armazena informações sobre um ponto de *commit*. Veja detalhes na próxima seção.
- `write.lock` : previne a abertura de múltiplas instâncias de `IndexWriter` no mesmo índice.
- `Segment Info (.si)` : armazena metadados sobre um segmento.
- `Compound File (.cfs, .cfe)` : arquivo opcional, consiste de um pequeno índice com a composição dos arquivos do índice principal.
- `Fields (.fnm, .fdx, .fdt)` : campos armazenados nos documentos.
- `Term Dictionary (.tim, .tip)` : armazena informações sobre os termos indexados, como ponteiros e dados sobre a proximidade

entre palavras.

- `Frequências (.doc)` : lista de documentos que contêm cada termo e sua frequência (quantidade de vezes em que ele aparece).
- `Positions (.pos)` : posição em que os termos ocorrem nos documentos.
- `Payloads (.pay)` : informação adicional das posições dos termos, como a posição inicial e final de cada palavra.
- `Normalization factors (.nvd, .nvm)` : fator de elevação para documentos e campos. Este valor é armazenado para ordenar o resultado da busca.
- `Per-Document Values (.dvd, .dvm)` : lista de documentos carregada na memória principal para acesso rápido,
- `Term Vectors (.tvx, .tvd, .tvf)` : um vetor de termos consiste do termo e da sua frequência.
- `Live Documents (.liv)` : quais arquivos são válidos.
- `Point values (.dii, .dim)` : armazena campos indexados como pontos, para busca numérica, inteiros longos e intersecção geométrica (2D e 3D).

A lista completa de tipos de arquivo do Lucene é esta que segue e também está disponível neste link: http://lucene.apache.org/core/7_4_0/core/org/apache/lucene/codecs/lucene70/package-summary.html/. Esses arquivos formam a estrutura do índice e têm funções bastante especializadas.

Segmentos

O índice é subdividido em fragmentos chamados de *segments*. Imaginando um índice com 60 MB de tamanho, teríamos 3 segmentos com aproximadamente 16 MB cada (esse é o padrão). Essa estratégia de usar segmentos é interessante porque o Lucene suporta índices com um grande número de documentos, e fragmentá-los em diversos arquivos é mais eficiente que gravar tudo em um único arquivo de tamanho indefinido.

Um índice com bilhões de documentos poderia gerar um arquivo de vários gigabytes, o que é ruim para a performance. Por isso, é mais

eficiente gerar vários arquivos pequenos, de tamanho controlado, para particionar o índice e garantir a velocidade de consulta.

Essa fragmentação é o que garante a boa performance do Lucene e está sempre em desenvolvimento, acompanhando a evolução do Java e do hardware. Por esse motivo, o Lucene constantemente muda a estrutura do índice, adicionando novos tipos de arquivo para aumentar o particionamento e velocidade. Deve-se lembrar que o volume de dados está cada vez maior e isso é o que motiva a popularidade da área de *big data*. Claro que não é essencial saber essa parte para implementar uma solução com Lucene, mas entender o funcionamento interno pode ser útil para modificações e personalizações.

Um índice típico contém vários arquivos, como visto na tabela anterior. Veja a próxima listagem com os arquivos gerados pelo primeiro teste com os dados da Wikipedia. Como são muitos, vamos mostrar apenas os primeiros.

```
-rw-r--r-- 1 marco marco       341 Oct  2 09:14 _g.cfe
-rw-r--r-- 1 marco marco  17995927 Oct  2 09:14 _g.cfs
-rw-r--r-- 1 marco marco       380 Oct  2 09:14 _g.si
-rw-r--r-- 1 marco marco       341 Oct  2 09:15 _h.cfe
-rw-r--r-- 1 marco marco  19929008 Oct  2 09:15 _h.cfs
-rw-r--r-- 1 marco marco       380 Oct  2 09:15 _h.si
-rw-r--r-- 1 marco marco       341 Oct  2 09:16 _i.cfe
-rw-r--r-- 1 marco marco   6789004 Oct  2 09:16 _i.cfs
-rw-r--r-- 1 marco marco       380 Oct  2 09:16 _i.si
-rw-r--r-- 1 marco marco       657 Oct  2 09:16 segments_3
```

Podemos identificar os segmentos pelo prefixo e o tipo de arquivo pela extensão. No nosso exemplo, são 3 segmentos verificados pelos prefixos `_g` , `_h` e `_i` . O prefixo é alfanumérico, ou seja, pode ser um número ou uma letra. Note que os *compound files*, aqueles com extensão *cfs*, têm aproximadamente o mesmo tamanho.

O indexador, por padrão, cria esses arquivos com tamanho parecido, enquanto o último arquivo pode ficar ligeiramente menor.

Essas configurações podem ser modificadas, como será visto nas próximas seções.

6.4 Controlando a segmentação do índice

O Lucene oferece ajustes avançados que podem ajudar a otimizar o índice. Essa otimização deve levar em conta um equilíbrio entre a indexação e a busca:

1. *Compound files*: particionamento do índice em arquivos;
2. *RAM buffer size*: a criação dos segmentos é baseada na memória;
3. *Max buffered docs*: a criação dos segmentos é baseada no número de documentos;
4. *Merge policy*: estratégia usada para juntar arquivos à medida que novos documentos são adicionados e removidos.

Compound files

Os *compound files* são estruturas usadas pelo Lucene para dividir o índice em partes menores e agilizar a busca. Assim, são partes ou subdivisões do índice principal. Novamente, considerando um índice de 60 MB, teremos 3 *compound files*, cada um com 16 MB (o tamanho padrão). A extensão para eles é *cfs*.

O uso de *compound files* é opcional. Com *compound files*, o número de arquivos do índice é reduzido, mas essa otimização custa tempo de processamento. Desabilitar a criação de *compound files* economiza tempo da CPU, mas são criados mais arquivos no disco. Em índices muito grandes, desabilitar os *compound files* pode diminuir o tempo de indexação.

Para testar os efeitos do uso dos *compound files* vamos alterar a classe `IndexadorArquivosLocais.inicializar` e simplesmente desabilitar

sua criação com a sintaxe `conf.setUseCompoundFile(false)`. Dessa forma:

```
    public void inicializar() throws IOException {
        // {...}
        IndexWriterConfig conf = new IndexWriterConfig(analyzer);
        conf.setUseCompoundFile(false);
        writer = new IndexWriter(diretorio, conf);
        // {...}
    }
```

Se indexarmos novamente o mesmo diretório de teste, o resultado será:

- Quantidade de arquivos: 89.996.
- Quantidade de bytes indexados: 521 MB.
- Tamanho do diretório do índice: 1.1 GB.
- Quantidade de arquivos no índice: 206.
- Tempo de indexação: 172 segundos.

Veja que o tempo de indexação foi bem parecido, mas a quantidade de arquivos no índice aumentou bastante, sendo quase o dobro. Abra o diretório do índice e veja que não há mais arquivos com a extensão *cfs*. Na listagem a seguir vamos mostrar apenas alguns dos arquivos do novo índice:

```
-rw-r--r--  1 marco marco       71 Jan  9 10:16 _10.dii
-rw-r--r--  1 marco marco    73257 Jan  9 10:16 _10.dim
-rw-r--r--  1 marco marco 63323076 Jan  9 10:16 _10.fdt
-rw-r--r--  1 marco marco    15188 Jan  9 10:16 _10.fdx
-rw-r--r--  1 marco marco      658 Jan  9 10:16 _10.fnm
-rw-r--r--  1 marco marco  6184240 Jan  9 10:16 _10_Lucene50_0.doc
-rw-r--r--  1 marco marco 15264528 Jan  9 10:16 _10_Lucene50_0.pos
-rw-r--r--  1 marco marco 67829392 Jan  9 10:16 _10_Lucene50_0.tim
-rw-r--r--  1 marco marco   125785 Jan  9 10:16 _10_Lucene50_0.tip
-rw-r--r--  1 marco marco    12329 Jan  9 10:16 _10.nvd
-rw-r--r--  1 marco marco       88 Jan  9 10:16 _10.nvm
-rw-r--r--  1 marco marco      521 Jan  9 10:16 _10.si
```

Este índice contém os mesmos documentos, mas veja que os segmentos têm outros tipos de arquivos. Isso acontece porque tiramos o *cfs*. Para concluir esse assunto, podemos entender que o *compound files* é uma opção que deve ser avaliada quando o índice é muito grande, da ordem de centenas de gigabytes, uma vez que índices pequenos, com poucos gigabytes, não percebem diferença de performance com essas alternativas de configuração.

RAM buffer size

A quantidade e o tamanho dos arquivos do índice são configurações que podem ser alteradas para otimizar o buscador. Podemos optar por (i) vários arquivos pequenos para otimizar a busca ou (ii) poucos arquivos grandes para otimizar a indexação. A primeira opção é a padrão, pois considera arquivos de aproximadamente 16 MB, que são adequados para a maioria das situações. A segunda opção é aumentar o tamanho dos arquivos do índice para otimizar a indexação, o que pode ser feito com a configuração do RAM *buffer size*. O *buffer* determina a quantidade de RAM usada para armazenar documentos antes de gravar um arquivo. Aumentar esse valor tende a aumentar a velocidade da indexação, por outro lado pode diminuir a performance da busca. O valor padrão é 16 MB e, no primeiro teste, vamos aumentar para 48 MB, isto é, 3 vezes o tamanho inicial. O objetivo é que tenhamos um índice com 3 vezes menos arquivos do que o padrão. Veja como fica no código:

```
// {...}
IndexWriterConfig conf = new IndexWriterConfig(analyzer);
conf.setRAMBufferSizeMB(48);
writer = new IndexWriter(diretorio, conf);
// {...}
```

Se criarmos novamente o índice com a nova configuração teremos apenas 49 arquivos no índice. Neste caso, a indexação tem uma pequena melhora de performance, como pode ser visto na lista:

- Quantidade de arquivos: 89.996.
- Quantidade de bytes indexados: 521 MB.

- Tamanho do diretório do índice: 1.1 GB.
- Quantidade de arquivos no índice: 49.
- Tempo de indexação: 165 segundos.

Max buffered docs

Determina a quantidade de documentos indexados na memória antes de gravá-los em um novo segmento. Assim como o *RAM buffer size*, aumentar esse número tem o potencial de acelerar a indexação para índices muito grandes. Por outro lado, a busca tende a ser mais lenta porque os arquivos do índice ficam maiores.

Esta configuração fica desabilitada por padrão. Para habilitá-la, defina um valor maior que 0. Nosso teste será com 30.000, ou seja, vamos armazenar 30.000 documentos na memória antes de gravar um novo segmento no disco. Para evitar o *flush* pela memória, temos de desabilitar o `RAMBufferSize`, como no código a seguir:

```
// {...}
IndexWriterConfig conf = new IndexWriterConfig(analyzer);
conf.setMaxBufferedDocs(30000);
conf.setRAMBufferSizeMB(
        IndexWriterConfig.DISABLE_AUTO_FLUSH);
writer = new IndexWriter(diretorio, conf);
// {...}
```

Resultado:

- Quantidade de arquivos: 89.996;
- Quantidade de bytes indexados: 521 MB;
- Tamanho do diretório do índice: 1.1 GB;
- Quantidade de arquivos no índice: 13;
- Tempo de indexação: 186 segundos.

Como temos cerca de 90.000 documentos e nosso *buffer* é de 30.000, o índice terá aproximadamente 3 segmentos em razão da configuração que fizemos. Veja o índice construído:

```
-rw-r--r-- 1 marco marco       341 Jan  9 12:04 _0.cfe
-rw-r--r-- 1 marco marco 257272845 Jan  9 12:04 _0.cfs
-rw-r--r-- 1 marco marco       364 Jan  9 12:04 _0.si
-rw-r--r-- 1 marco marco      3811 Jan  9 12:05 _1_1.liv
-rw-r--r-- 1 marco marco       341 Jan  9 12:05 _1.cfe
-rw-r--r-- 1 marco marco 349555704 Jan  9 12:05 _1.cfs
-rw-r--r-- 1 marco marco       364 Jan  9 12:05 _1.si
-rw-r--r-- 1 marco marco      3811 Jan  9 12:06 _2_1.liv
-rw-r--r-- 1 marco marco       341 Jan  9 12:06 _2.cfe
-rw-r--r-- 1 marco marco 462721471 Jan  9 12:06 _2.cfs
-rw-r--r-- 1 marco marco       364 Jan  9 12:06 _2.si
-rw-r--r-- 1 marco marco       260 Jan  9 12:06 segments_1
-rw-r--r-- 1 marco marco         0 Jan  9 12:03 write.lock
```

O valor do *max buffered docs*, assim como o *RAM buffer size*, são ajustes finos que devem ser feitos apenas em sistemas grandes, com dezenas de milhões de documentos. Menos do que isso não faz muita diferença na performance.

Merge policy

Vimos configurações para construção do índice a partir do zero. Mas em um sistema de busca real que está em produção o índice não pode ser criado e recriado frequentemente. Assim, durante a operação do sistema, novos itens são incluídos e excluídos, de forma que os arquivos do índice (os segmentos) precisam ser atualizados para refletir as alterações.

O recurso do Lucene para essa tarefa é o *merge policy*, ou política de junção. Essa estratégia é usada quando há mudanças nos segmentos do índice, isto é, quando há inclusão ou exclusão de novos documentos. O tamanho dos segmentos é gerenciado pela política, então, durante sua alteração, um ou vários segmentos são agrupados e formam outros segmentos.

Cada vez que o índice é alterado, o `IndexWriter` verifica qual foi a implementação de *merge policy* selecionada. Cada política é implementada em uma subclasse de `MergePolicy` e define parâmetros para o algoritmo de junção de segmentos, como a

quantidade mínima e máxima de memória ou de documentos antes de gravar os dados no disco. As 3 políticas do Lucene estão listadas logo a seguir. O processo de junção funciona assim:

- Ordenação dos segmentos por nome;
- Agrupamento dos segmentos em grupos de segmentos, chamados de níveis;
- Para cada nível, verificar quais segmentos serão juntados em um único segmento.

Na figura, pode-se ver como isso acontece. O algoritmo agrupou os segmentos em 2 níveis. Normalmente isso é feito de acordo com o tamanho de cada segmento. Depois, o `MergePolicy` verifica como será feita a fusão. Em alguns casos, vários segmentos são agrupados e geram um novo segmento que será gravado no disco. Quando o segmento é muito grande, ele não é mesclado com nenhum outro.

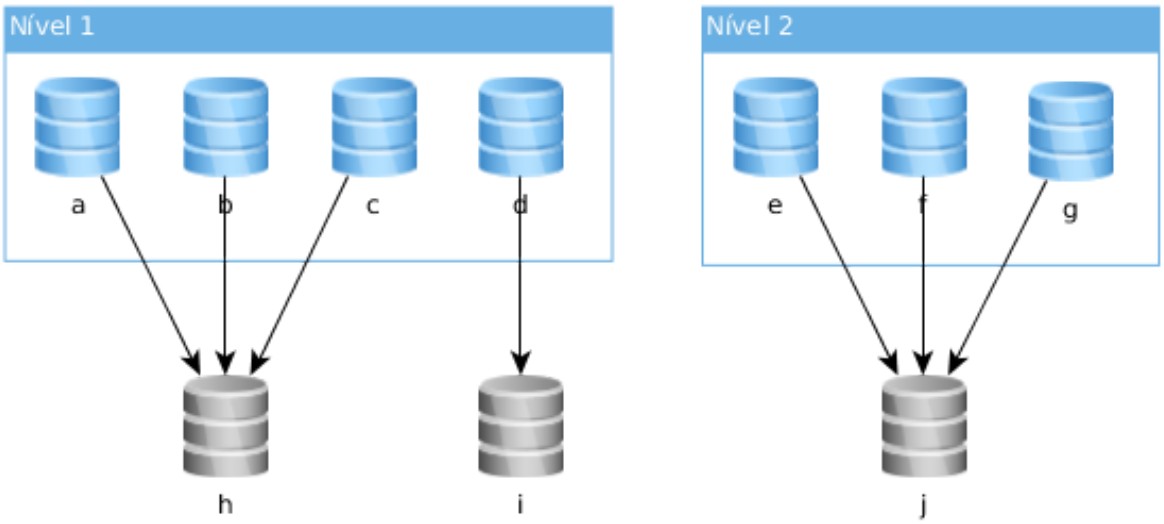


Figura 6.4: Merge policy

A mudança nos valores dos parâmetros pode diminuir ou aumentar as operações de leitura e escrita no disco. O padrão é que os parâmetros estejam equilibrados para as operações de indexação e busca, mas pode-se escolher reservar mais memória para um lado

ou outro. Este artigo do Elasticsearch fala um pouco mais sobre o funcionamento da junção: https://www.elastic.co/guide/en/elasticsearch/reference/current/index-modules-merge.html/.

A política padrão é a `TieredMergePolicy`, que tenta criar segmentos aproximadamente do mesmo tamanho. O tamanho máximo de um segmento produzido por essa política é de aproximados 5 GB. Índices maiores que isso podem apresentar lentidão no momento da fusão dos arquivos durante o processo de escrita no disco. Quando os segmentos são juntados, um novo arquivo é criado. Se você tiver 5 GB para copiar podemos considerar que será uma operação pesada para o computador.

Para escolher uma *merge policy* faça como na sintaxe a seguir. Essa já é a política padrão, então, o código a seguir é redundante. Na sequência, veremos as outras políticas e como elas se comportam.

```
IndexWriterConfig conf = new IndexWriterConfig(analyzer);
TieredMergePolicy mergePolicy = new TieredMergePolicy();
conf.setMergePolicy(mergePolicy);
writer = new IndexWriter(diretorio, conf);
```

Os valores iniciais dessa *merge policy* estão no bloco de código a seguir. Os valores que merecem menção são `maxMergedSegmentMB` e `segmentsPerTier`. Eles representam o tamanho máximo de segmentos que podem ser agrupados e a quantidade de segmentos em um nível, respectivamente.

```
mergePolicy=[TieredMergePolicy: maxMergeAtOnce=10,
maxMergeAtOnceExplicit=30, maxMergedSegmentMB=5120.0, floorSegmentMB=2.0,
forceMergeDeletesPctAllowed=10.0, segmentsPerTier=10.0,
maxCFSSegmentSizeMB=8.796093022207999E12, noCFSRatio=0.1
```

A política `LogByteSizeMergePolicy` determina a frequência com que os segmentos são juntados através do fator de fusão (*merge factor*) que define, entre outras coisas, quantos segmentos serão agrupados para formar um novo segmento.

Valores menores que 10 (o padrão) usam menos RAM para indexação, liberando essa memória para a busca que tende a ficar mais rápida, isso considerando que os usuários estão consultando com o sistema normalmente enquanto novos documentos são incluídos. Valores maiores que 10 são adequados para a construção do índice quando o sistema não está sendo usado para buscas. No código a seguir o valor está configurado para 40. Veja no código como fica:

```
IndexWriterConfig conf = new IndexWriterConfig(analyzer);
LogByteSizeMergePolicy mergePolicy =
        new LogByteSizeMergePolicy();
mergePolicy.setMergeFactor(40);
conf.setMergePolicy(mergePolicy);
writer = new IndexWriter(diretorio, conf);
```

Os valores dessa *merge policy* estão a seguir. Veja que o `mergeFactor` é 40.

```
mergePolicy=[LogByteSizeMergePolicy: minMergeSize=1677721, mergeFactor=40,
maxMergeSize=2147483648, maxMergeSizeForForcedMerge=9223372036854775807,
calibrateSizeByDeletes=true, maxMergeDocs=2147483647,
maxCFSSegmentSizeMB=8.796093022207999E12, noCFSRatio=0.1]
```

A última política a ser analisada é a `LogDocMergePolicy` . Ela funciona de forma parecida com a anterior, contudo, utiliza o número de documentos de um segmento como medida padrão e não a quantidade de memória. Para empregar essa política, use o código:

```
IndexWriterConfig conf = new IndexWriterConfig(analyzer);
LogDocMergePolicy mergePolicy = new LogDocMergePolicy();
conf.setMergePolicy(mergePolicy);
writer = new IndexWriter(diretorio, conf);
```

Os valores iniciais são estes que seguem:

```
mergePolicy=[LogDocMergePolicy: minMergeSize=1000, mergeFactor=10,
maxMergeSize=9223372036854775807,
maxMergeSizeForForcedMerge=9223372036854775807,
```

```
calibrateSizeByDeletes=true, maxMergeDocs=2147483647,
maxCFSSegmentSizeMB=8.796093022207999E12, noCFSRatio=0.1]
```

Considerações sobre otimização

A indexação e a busca são sensíveis ao hardware utilizado. Memória, processador e discos ditam como será a performance da aplicação. Especialmente no caso da busca, use toda a RAM disponível. Dessa forma, o buscador vai armazenar os documentos na memória, aumentando a performance. Por isso é recomendado o uso dos parâmetros de JVM, como `-Xmx8g -Xms8g -server`, se você tiver 8 GB disponíveis.

Por outro lado, a indexação é uma atividade que depende muito do mecanismo de IO do hardware. Discos mais novos são geralmente mais rápidos e isso se reflete na velocidade da indexação. E ainda existe a opção de usar um disco SSD (*solid-state-disk*), um recurso mais caro, porém mais eficiente se comparado aos discos rígidos tradicionais. Um SSD pode ser 5 vezes mais rápido que um disco tradicional (HDD), mas custa até 5 vezes mais caro.

6.5 Analyzer

Até aqui utilizamos o `StandandAnalyzer`, que é o analisador padrão do Lucene. Um analisador é o tradutor entre o texto original e o que será indexado. Ele é responsável por transformar o texto original em fragmentos que serão indexados. Vale citar que o `StandardAnalyser` foi escrito para a língua inglesa. Existem outros analisadores, inclusive para o português.

Cada analisador faz uma série de transformações no texto, antes ou depois da indexação. Antes de ser indexado, o texto passa por um processo de análise (pré-processamento) onde é convertido em pequenas partes, ou termos. Esses termos serão utilizados posteriormente durante a busca por meio do índice. Os documentos

que contêm os mesmos termos que a busca do usuário serão recuperados pelo Lucene.

Os analisadores são implementações da classe abstrata `org.apache.lucene.analysis.Analyzer`. O Lucene conta com alguns já implementados e cada um processa o texto de forma diferente. Vejamos alguns analisadores comuns e suas características:

Analisador	Descrição
`StandardAnalyzer`	Tipo mais genérico de analisador, transforma as letras em minúsculas, retira a pontuação e retira *stop words* em inglês.
`SimpleAnalyzer`	Transforma as letras em minúsculas e retira a pontuação.
`BrazilianAnalyzer`	Transforma as letras em minúsculas, retira a pontuação, retira acentos, usa a lista de *stop words* em português e faz a radicalização (*stemming*) de cada termo. Por exemplo, o radical de "origem" e "original" é "orig".
`WhitespaceAnalyzer`	Separa os termos pelo espaço entre as palavras.
`KeywordAnalyzer`	Considera o texto inteiro como um único termo. Pode usado para IDs e CEPs.

A análise de um texto envolve uma área da computação chamada de Processamento de Linguagem Natural (PLN), que tenta traduzir o conteúdo de forma que o computador possa entender seu sentido, ou seja, o PLN é um campo da computação que tenta descrever a linguagem humana em termos computacionais.

Dada a complexidade da linguagem, e especialmente a do português, é necessário executar algumas operações sobre o texto para que ele seja interpretado pelo sistema de busca. A primeira

técnica que veremos é a normalização e na sequência veremos os exemplos com código.

Normalização é a medida de importância de um termo dentro do texto. Esse cálculo é feito automaticamente pelo Lucene usando a fórmula `1/raiz(numTermos)`. Para sistemas de PLN mais precisos, pode-se criar uma fórmula personalizada.

Stemming (extração da raiz da palavra) é uma técnica linguística para transformar uma palavra em seu elemento originário e irredutível, que contém apenas o seu significado. Por exemplo, os termos *programação*, *programador* e *programadora* são convertidos para a raiz *program*. Com essa técnica, o Lucene procura a partir da raiz (*program*) e não a partir dos termos. Isso é interessante porque você pode procurar por *programador* e o Lucene vai encontrar todos os documentos que contém a raiz *program*, incluindo *programadora*, *programadores* etc.

Com o *stemming*, o Lucene faz a remoção dos afixos das palavras, que são os prefixos e sufixos. Prefixo é aquele termo que fica antes da raiz. Por exemplo, "dizer" e "contradizer"; "tensão" e "hipertensão". O sufixo fica após a raiz, como em "dente" e "dentista"; "ferro" e "ferragem".

A **retirada de *stop words* e da pontuação** é a última técnica adotada pelo Lucene para os processos de indexação e busca. As *stop words* foram discutidas no capítulo *2. Conceitos de recuperação da informação*, e são aquelas palavras e símbolos de baixa relevância dentro do texto, como preposições e artigos (a, o, de, da, do). Essas palavras não são importantes em um sistema de busca porque você não vai conseguir encontrar um texto pesquisando apenas por uma delas. A pontuação (ponto, vírgula, exclamação) e símbolos especiais (`%` , `@` , `$`) são removidos por padrão. Geralmente, não precisamos destes símbolos, mas se existir algum caso especial no seu sistema, então você tem de personalizar seu processo de análise, como veremos no capítulo *9. Recursos avançados*.

Observação: a lista de *stop words* é definida pelo analisador que você escolher. Se escolher o `StandardAnalyzer`, as *stop words* são em Inglês (in, on, at, the). As *stop words* em português estão no `BrazilianAnalyzer`, que será apresentado mais tarde neste capítulo.

Normalização

A normalização é uma técnica usada para tentar diminuir a diferença entre os tamanhos dos textos de um índice. Em um índice, pode aparecer um texto com poucas palavras e outro com centenas de páginas. Neste caso, o resultado da busca pode ser afetado.

Matematicamente falando, a normalização é uma equação genérica para equilibrar o peso dos termos de um texto. É para ponderar esses casos que usamos a normalização. Tentamos equilibrar as duas situações.

Agora vamos imaginar uma situação onde uma palavra aparece em um texto pequeno. Ora, se o texto é pequeno, cada palavra é importante. Ao contrário, em um texto grande, cada palavra teria um valor pequeno. É o mesmo caso de uma pessoa em uma equipe de trabalho e uma pessoa no metrô lotado. Na equipe a pessoa é muito importante, mas no metrô ela passa despercebida. Veremos um exemplo prático com três documentos que falam sobre Linux.

Documento	Texto
0	Geralmente você pode usar as opções 'mute' ou 'unmute' para gerenciar os drivers ALSA no Linux
1	As versões 0.3 e 0.4 têm vários problemas no linux devido à reestruturação da interface do mixer que teve de ser reescrito em função de problemas identificados anteriormente no linux
2	Você precisa carregar o módulo para o seu cartão de som ou usar o utilitário 'kmod' do linux

A classe a seguir, que reutiliza a `IndiceEmMemoria`, mostra como o Lucene normaliza os campos. Existe um método exatamente para isso, o `IndexSearcher.explain`, que retorna uma explanação sobre como o documento foi analisado. A ideia é pesquisar a palavra `linux` entre os documentos.

```
public class TesteExplicacaoConsulta {
    private static final Logger logger = Logger
            .getLogger(AnalisadorDeTermos.class);

    @Test
    public void testeExplicacao()
            throws IOException, ParseException {
        Directory diretorio = new IndiceEmMemoria()
                .getRamDirectory();
        IndexReader reader = DirectoryReader.open(diretorio);
        IndexSearcher searcher = new IndexSearcher(reader);
        QueryParser parser = new QueryParser("",
                new StandardAnalyzer());
        String consulta = "conteudo:linux";
        Query query = parser.parse(consulta);
        TopDocs docs = searcher.search(query, 3);
        for (int i = 0; i < docs.scoreDocs.length; i++) {
            Explanation explain = searcher.explain(query,
                    docs.scoreDocs[i].doc);
            logger.info(explain);
        }
        diretorio.close();
        reader.close();
    }
}
```

O resultado está listado a seguir. Com base nele podemos entender como o Lucene ordena o resultado, considerando a normalização e outros parâmetros já vistos, como o TF-IDF. Veja:

```
INFO  0.16211805 = weight(conteudo:linux in 1) [BM25Similarity], result
of:
  0.16211805 = score(doc=1,freq=2.0 = termFreq=2.0
), product of:
```

```
    0.13353139 = idf(docFreq=3, docCount=3)
    1.2140819 = tfNorm, computed from:
      2.0 = termFreq=2.0
      1.2 = parameter k1
      0.75 = parameter b
      19.333334 = avgFieldLength
      28.444445 = fieldLength

INFO  0.14366448 = weight(conteudo:linux in 0) [BM25Similarity], result
of:
  0.14366448 = score(doc=0,freq=1.0 = termFreq=1.0
), product of:
    0.13353139 = idf(docFreq=3, docCount=3)
    1.0758854 = tfNorm, computed from:
      1.0 = termFreq=1.0
      1.2 = parameter k1
      0.75 = parameter b
      19.333334 = avgFieldLength
      16.0 = fieldLength

INFO  0.12925221 = weight(conteudo:linux in 2) [BM25Similarity], result
of:
  0.12925221 = score(doc=2,freq=1.0 = termFreq=1.0
), product of:
    0.13353139 = idf(docFreq=3, docCount=3)
    0.96795374 = tfNorm, computed from:
      1.0 = termFreq=1.0
      1.2 = parameter k1
      0.75 = parameter b
      19.333334 = avgFieldLength
      20.897959 = fieldLength
```

O resultado é meio intimidador, contudo, se olhar com cuidado vai perceber que não é tão complicado. Em todos os documentos há ao menos uma ocorrência da palavra `linux` e a ordem do resultado é 1, 0 e 2. Significa que a palavra `linux` é mais importante no documento 1, depois no documento 0 e, por fim, no documento 2. Essa ordenação foi definida pela normalização.

Para facilitar a visualização, eu criei a tabela a seguir com os valores calculados pelo Lucene.

Campo	Documento 0	Documento 1	Documento 2
Nota	0.14366448	0.16211805	0.12925221
tfNorm	1.0758854	1.2140819	0.96795374
idf	0.13353139	0.13353139	0.13353139
termFreq	1.0	2.0	1.0
avgFieldLength	19.333334	19.333334	19.333334
fieldLength	16.0	28.444445	20.897959

Onde:

- `Nota` (score): é a pontuação final do documento para esta consulta;
- `tfNorm` : valor da normalização do campo;
- `idf` : valor do *inverse document frequency*, visto no capítulo *2. Conceitos*;
- `termFreq` : quantas vezes o termo aparece no campo;
- `avgFieldLength` : tamanho médio do campo pesquisado;
- `fieldLength` : quantidade total de termos do campo.

As notas foram calculadas pela classe `BM25Similarity` e consideram a frequência do termo pesquisado no documento bem como a normalização e o tamanho médio do campo pesquisado. As equações podem ser conferidas neste link: https://lucene.apache.org/core/7_4_0/core/org/apache/lucene/search/similarities/BM25Similarity.html/.

Stemming

O Lucene funciona bem para a função de analisar palavras, montar um índice invertido e consultar com velocidade os documentos correspondentes. Em síntese, o Lucene é eficiente para recuperar itens do índice através da combinação entre os termos da consulta e os termos indexados nos documentos.

Logo a seguir temos um utilitário para ver como o `Analyzer` processa as palavras do texto. Note que substituímos o espaço em branco entre as palavras pelo caractere *pipe* (`|`) para a visualização ficar clara. O resultado é a lista de palavras que estão sendo consideradas para montar o índice.

A classe `AnalisadorDeTermos` implementa uma funcionalidade interessante, que mostra como o Lucene interpreta as palavras por meio dos analisadores. A análise do texto mostra cada um dos termos do texto e o resultado das transformações.

```
public class AnalisadorDeTermos {
    private static final Logger logger = Logger
            .getLogger(AnalisadorDeTermos.class);

    public static void analisarFrase(Analyzer analyzer,
            String texto) {
        try {
            TokenStream stream = analyzer.tokenStream(null,
                    new StringReader(texto));
            stream.reset();
            StringBuilder termos = new StringBuilder();
            termos.append(analyzer.getClass().getSimpleName());
            termos.append(" => ");
            while (stream.incrementToken()) {
                Iterator<AttributeImpl> ite = stream
                        .getAttributeImplsIterator();
                AttributeImpl impl = ite.next();
                termos.append(impl);
                termos.append("|");
            }
            logger.info(termos);
        } catch (IOException e) {
```

```
            logger.error(e);
        }
    }
}
```

Para testar vamos usar a classe a seguir. Perceba que há uma frase inicial que pode ser alterada para ilustrar outras situações. O objetivo é imprimir como o analisador processou a frase.

```
public class AnalisadorDeTermosTest {
    private String frase = "De origem humilde até a riqueza: "
            + "veja 11 bilionários que eram "
            + "pobres na infância.\n"
            + "Trabalho duro e resiliência é a "
            + "característica comum a todos. "
            + "Contudo, eles representam apenas "
            + "1% da população.";

    @Test
    public void analisar() throws IOException {
        Analyzer standardAnalyzer = new StandardAnalyzer();
        AnalisadorDeTermos.analisarFrase(standardAnalyzer,
                frase);
        Analyzer simpleAnalyzer = new SimpleAnalyzer();
        AnalisadorDeTermos.analisarFrase(simpleAnalyzer, frase);
        Analyzer brazilianAnalyzer = new BrazilianAnalyzer();
        AnalisadorDeTermos.analisarFrase(brazilianAnalyzer,
                frase);
        Analyzer whiteSpaceAnalyzer = new WhitespaceAnalyzer();
        AnalisadorDeTermos.analisarFrase(whiteSpaceAnalyzer,
                frase);
        Analyzer keyWordAnalyzer = new KeywordAnalyzer();
        AnalisadorDeTermos.analisarFrase(keyWordAnalyzer,
                frase);
    }
    // {...}
}
```

O resultado da execução dessa classe é mostrado no *log* que está logo a seguir. Veja que cada analisador processa os dados de uma forma diferente. Novamente, a escolha fica por conta do tipo de

busca que sua aplicação precisa. Note como o `BrazilianAnalyzer` interpreta as palavras em português, incluindo a questão das *stop words* e *stemming*.

```
INFO  StandardAnalyzer =>
de|origem|humilde|até|riqueza|veja|11|bilionários|que|eram|pobres|na|infân
cia|trabalho|duro|e|resiliência|
é|característica|comum|todos|contudo|eles|representam|apenas|1|da|populaçã
o|
INFO  SimpleAnalyzer =>
de|origem|humilde|até|a|riqueza|veja|bilionários|que|eram|pobres|na|infânc
ia|trabalho|duro|e|resiliência|
é|a|característica|comum|a|todos|contudo|eles|representam|apenas|da|popula
ção|
INFO  BrazilianAnalyzer =>
orig|humild|ate|riquez|vej|11|bilionari|eram|pobr|infanc|trabalh|dur|resil
iente|é|caracterist|comum|represent|apen|1|popul|
INFO  WhitespaceAnalyzer =>
De|origem|humilde|até|a|riqueza:|veja|11|bilionários|que|eram|pobres|na|in
fância.|Trabalho|duro|e|resiliência|
é|a|característica|comum|a|todos.|Contudo,|eles|representam|apenas|1%|da|p
opulação.|
INFO  KeywordAnalyzer => De origem humilde até a riqueza: veja 11
bilionários que eram pobres na infância.
Trabalho duro e resiliência é a característica comum a todos. Contudo,
eles representam apenas 1% da população.|
```

Durante a indexação e busca é importante utilizar o mesmo `Analyzer` para que o usuário consiga encontrar os documentos com precisão. Se utilizar um analisador para indexar e outro diferente para a busca, o resultado é imprevisível e, na melhor das situações, não tem precisão garantida.

Stop words

Podemos escolher as *stop words* da nossa aplicação através de um parâmetro no método construtor do `StandardAnalyzer` . Por padrão, o Lucene usa as *stop words* em inglês, o que pode não ser adequado para nossa aplicação.

A seguir, podemos ver uma lista limitada com algumas das *stop words* que o Lucene utiliza no `BrazilianAnalyzer`. Cada `Analyzer` tem sua própria lista de *stop words*. O Lucene tem analisadores para diversas línguas, como espanhol, francês, chinês, italiano etc.

- a, ainda, alem, ambas, ambos, antes, ao, aonde, aos, apos, aquele, aqueles, as, assim;
- com, como, contra, contudo, cuja, cujas, cujo, cujos;
- da, das, de, dela, dele, deles, demais, depois, desde, desta, deste, dispoe, dispoem, diversa, diversas, diversos, do, dos, durante.

No caso de usar o `StandardAnalyzer`, que é um analisador genérico, podemos informar quais *stop words* vamos usar.

```
public void analisarComStopWords() throws IOException {
    // Cria lista de stop words em português
    Collection<String> listaDeStopWords =
            new ArrayList<String>();
    listaDeStopWords.add("de");
    listaDeStopWords.add("até");
    listaDeStopWords.add("que");
    listaDeStopWords.add("e");
    listaDeStopWords.add("a");
    CharArraySet stopWords =
            new CharArraySet(listaDeStopWords, true);
    // Aplica a lista ao StandardAnalyzer
    Analyzer standardAnalyzer =
            new StandardAnalyzer(stopWords);
    AnalisadorDeTermos.analisarFrase(standardAnalyzer,
            frase);
}
```

Verifique o resultado e compare com o anterior. Podemos ver que as *stop words* não estão presentes, como era esperado.

```
INFO  StandardAnalyzer =>
origem|humilde|riqueza|veja|11|bilionários|eram|pobres|na|infância|trabalh
o|duro|resiliência|
```

é|característica|comum|todos|contudo|eles|representam|apenas|1|da|populaçã
o|

Resumo

Neste capítulo vimos configurações avançadas e otimizações para a construção do índice e para as buscas. O conteúdo é um pouco mais extenso, mas também é um conhecimento útil quando o índice começa a crescer em tamanho ou volume de acessos. Essas são definições centralizadas no `IndexWriter` e no `IndexReader`, que foram explicados em detalhe.

É difícil entender como são feitos os cálculos, então, para clarear um pouco as ideias, use o *debug* e o *explain*. Com esses recursos conseguimos ver o comportamento interno do índice. É uma alternativa avançada, porém, altamente recomendável, particularmente para quem pretende criar extensões ou personalizações do Lucene.

O índice é formado por segmentos, pequenos subíndices. Para controlar sua criação podemos alterar muitos parâmetros, incluindo os *compound files*, *buffer* e *merge policies*. Cada um desses parâmetros influencia na criação dos segmentos do índice. Um segmento é composto por vários arquivos e cada um armazena um tipo de informação diferente, não apenas o texto indexado, mas também a posição de cada palavra.

No entanto, nem todo índice gera segmentos. Durante testes pontuais use o `RAMDirectory`, um índice volátil que não deve ser usado em produção, mas que atende bem quando precisamos fazer experimentos.

Para descobrir os valores mais adequados da configuração para sua aplicação não tem mágica. Você precisa testar e avaliar os resultados com ferramentas específicas, como o JMeter e o VisualVM. O JMeter é uma ferramenta para teste de carga e

performance, enquanto o VisualVM é um *profiler* para análise de consumo da CPU e memória.

Este capítulo trouxe algumas novidades, como a indexação da Wikipedia, uma ótima fonte de dados textuais para pesquisa e, por consequência, para nossos testes de indexação. Outra curiosidade foi o uso da linguística durante o processo de radicalização (*stemming*) e *stop words* em português com o `BrazilianAnalyzer`.

Com o conteúdo apresentado até agora é possível fazer um bom uso das características do Lucene. O próximo passo será fazer a integração com sistemas corporativos. A ideia é usar inicialmente o Lucene embutido na aplicação, e depois por meio do *Hibernate Search*.

Capítulo 7
Integração com sistemas corporativos

O Lucene é uma biblioteca que pode ser facilmente acoplada a um sistema existente. Há diferentes formas de se fazer isso e neste capítulo veremos algumas dessas estratégias. Para simular um sistema de informação, foi criado um sistema de *e-commerce* fictício que implementa as principais funcionalidades deste tipo de aplicação.

Vamos utilizar o Lucene para fazer consultas de produtos simulando a navegação de um cliente no site, que pode estar buscando um item muito específico, navegando à procura de um presente qualquer ou para verificar o histórico de pedidos.

O projeto está disponível no GitHub (https://github.com/masreis/e-commerce) e utiliza as tecnologias mais recentes para construção de sistemas web na plataforma Java. São estas as bibliotecas e produtos utilizados:

- Lucene: biblioteca para busca textual
- Maven: gerenciador do *build* da aplicação
- JPA / Hibernate: persistência
- JSF / PrimeFaces: interfaces web
- MySQL: Sistema Gerenciador de Banco de Dados Relacional
- Tomcat: Servlet Container

A aplicação de exemplo conta com páginas web escritas em `xhtml` e PrimeFaces, mas vale lembrar que não é nosso objetivo entrar em detalhes sobre frameworks web. O ponto central do livro continua sendo o sistema de busca.

7.1 Modelo de dados

O sistema de e-commerce compreende poucas tabelas, contudo, conseguimos mostrar várias situações diferentes e complexas, como a questão da multiplicidade entre tabelas e no próprio JPA. A ideia é simular um sistema real, com um cenário que você provavelmente enfrentará.

As tabelas podem ser conferidas no modelo de dados disponível na imagem a seguir:

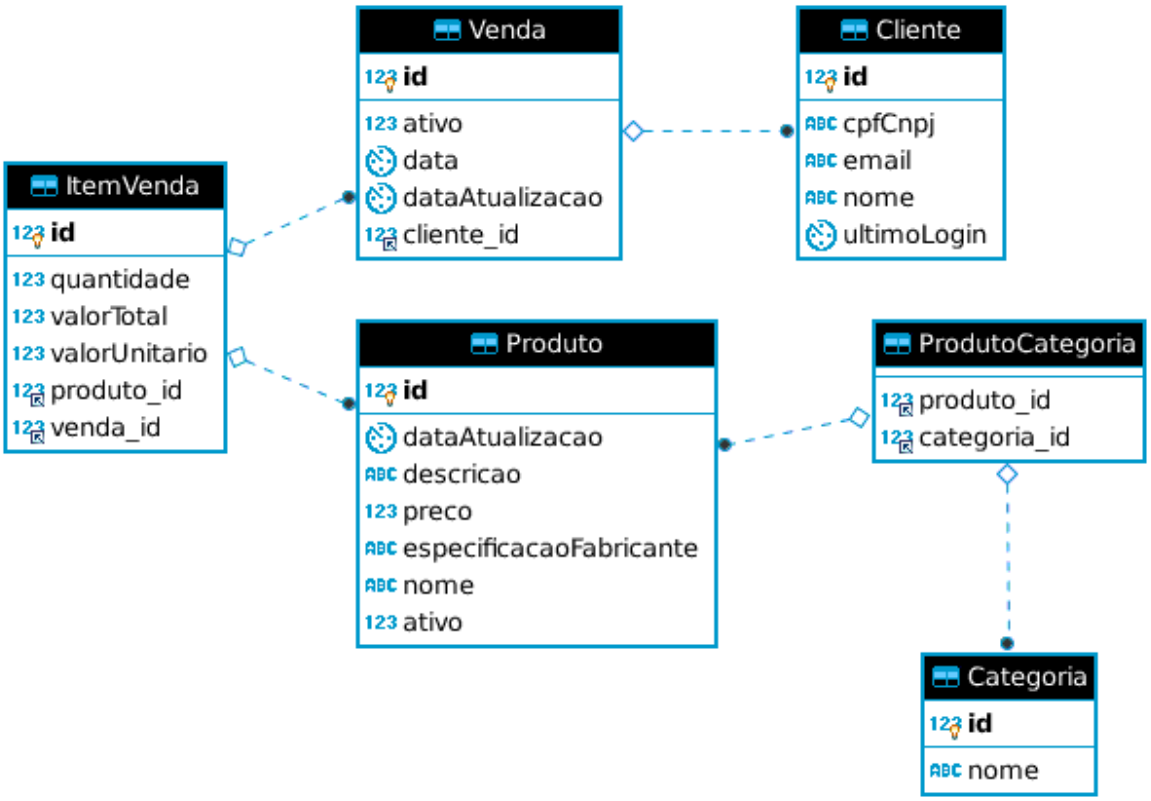


Figura 7.1: Modelo de dados

Tabela	Descrição
`Categoria`	Classificação do produto. Ex.: acessórios para celular, memória, computador.
`Cliente`	A pessoa que realiza a compra.

Tabela	Descrição
`ItemVenda`	Cada produto que compõe a venda.
`Produto`	Cada mercadoria vendida na loja.
`ProdutoCategoria`	Relacionamento N:M entre Produto e Categoria. Um produto pode ter várias categorias.
`Venda`	Registra as saídas dos produtos.

Temos menos classes de entidade que tabelas devido ao relacionamento `@ManyToMany` da classe `Produto`, já que ele pode pertencer a várias categorias. Por exemplo, um celular pode estar na categoria `Telefonia` e `Eletrônicos`. É uma complexidade que adicionamos para simular um sistema real, que tem esse tipo de detalhe.

A solução com o Lucene não leva em consideração as multiplicidades entre entidades. Um único documento do tipo *Produto* terá todos os dados que estão associados a ele, incluindo as categorias. Perceba que é diferente do modelo de dados relacional, em que cada um dos itens é armazenado em uma tabela diferente.

Os códigos das classes de entidade e outras questões de arquitetura não serão muito detalhados, uma vez que este não é o objetivo do livro. Vamos nos limitar a mostrar apenas o que for necessário para rodar todos os exemplos e aplicar os recursos do Lucene. Com isso, não vamos explicar conceitos de JPA como `NamedQuery`, `EntityManager`, *beans*, filtros etc. Não é um grande problema porque na prática você não vai precisar alterar a aplicação. Uma vez construída, os dados (ao menos o que queremos imediatamente) são carregados automaticamente por meio do Maven.

Neste capítulo vamos criar dois índices. O primeiro vai guardar os produtos e o segundo vai conter as vendas. É importante notar que já temos tabelas para guardar esses dados, mas o objetivo é exatamente não as usar, ou seja, o propósito é evitar fazer consultas no banco de dados relacional.

Outro ponto importante é que não há relação entre as tabelas do banco de dados e os índices do Lucene. O primeiro índice (`Produto`) é usado quando o cliente está procurando a mercadoria. Ele permite que busquemos por categoria, nome, descrição e especificação do produto. Por exemplo, o cliente começa pesquisando pela categoria `Telefonia` e depois filtra apenas os itens com preço entre R$ 500 e RS 600. O segundo índice (`Venda`) mostra o histórico de pedidos e pode ser usado tanto pelo cliente como pelo gerente do site. O cliente entra e vê a lista de todas as compras já feitas, bem como o gerente pode filtrar as vendas por qualquer um dos campos disponíveis no índice.

A classe `Categoria` associa um produto a um grupo, como `Eletrodomésticos` e `Telefonia` . O projeto tem algumas categorias de exemplo que são carregadas no banco de dados para efeito de teste. Para simplificar o código, não estão sendo mostrados os detalhes de implementação do JPA, como as anotações `@Entity` , `@Id` e `@ManyToOne` .

```
public class Categoria {
    private Long id;
    private String nome;
    // {...}
}
```

Um `Cliente` representa cada comprador do sistema. Para acessar o sistema, precisamos ao menos de um cliente cadastrado. Veja que não tem sequer senha, apenas e-mail. Novamente, a aplicação é apenas para teste.

```
public class Cliente {
    private Long id;
```

```
    private String email;
    private String nome;
    private Date ultimoLogin;
    // {...}
}
```

Um `ItemVenda` representa cada produto que compõe a venda. Do ponto de vista do banco de dados, um item está relacionado a um produto e a uma venda. Essas relações são importantes quando se modela um banco de dados relacional, mas não desaparecem quando modelamos um índice. Significa que cada item gravado no índice `Venda` terá todos os campos das tabelas `Cliente`, `ItemVenda`, `Produto` e `Venda`. Veremos como fica esse índice na próxima seção do capítulo.

```
public class ItemVenda {
    private Long id;
    private Produto produto;
    private BigDecimal valorUnitario;
    private BigDecimal valorTotal;
    private Integer quantidade;
    // {...}
}
```

`Produto` é um dos itens centrais da aplicação. Os artigos devem ser facilmente encontráveis no site e sabemos que qualquer dificuldade nesta busca gera perda de vendas. Por isso, a busca eficiente é indispensável nas aplicações modernas. O cliente tem de alcançar o produto com pouco esforço. O campo `especificacaoFabricante` é usado para guardar informações fornecidas pelo fabricante do produto, como o manual ou um arquivo com especificações.

```
public class Produto {
    private Long id;
    private String nome;
    private String descricao;
    private BigDecimal preco;
    private boolean ativo;
    private byte[] especificacaoFabricante;
    private Set<Categoria> categorias =
```

```
            new HashSet<Categoria>(0);
    private Date dataAtualizacao;
    // {...}
}
```

A `Venda` é o resultado da busca do produto. Quando o cliente finaliza a venda, podemos considerar que o sistema teve sucesso, mas o processo não termina aí. É importante que o pós-venda seja tão eficiente quanto a pré-venda. Se houver algum problema futuro, os pedidos devem ser também facilmente encontráveis. Com o índice `Venda` conseguiremos atender esse requisito para satisfação completa do cliente.

```
public class Venda {
    private Long id;
    private Date data;
    private Cliente cliente;
    private Date dataAtualizacao;
    private List<ItemVenda> itensVenda =
            new ArrayList<ItemVenda>();
    // {...}
```

7.2 Um buscador para sistemas web

Uma característica importante dos sistemas projetados para internet é a capacidade de suportar um grande volume de acessos simultâneos. O Lucene suporta essa propriedade, contudo, precisa de algumas adaptações, sendo que a mais importante é o compartilhamento de um único `IndexSearcher` entre os usuários do sistema. Essa técnica, vista nesta seção, visa a economizar recursos, porque podemos reutilizar a mesma instância sem acessar diversas vezes o sistema de arquivos.

Com essa abordagem, um servidor modesto é capaz de suportar uma maior quantidade de acessos. Sendo assim, criamos um novo

buscador para a aplicação web, com o recurso do compartilhamento do `IndexSearcher` para melhorar a performance.

Outra característica do novo buscador é que ele permite a utilização de mais de um índice (`Venda` e `Produto`) pelo sistema, além de otimizar a abertura do `IndexSearcher` . Na prática, poderíamos usar um único índice, mas a manutenção seria mais complexa. Com índices separados para cada tipo de informação, temos a vantagem de que o código será mais fácil de manter ao longo do tempo.

Para começarmos a implementação, vamos criar a enumeração `TipoIndice` , com os valores `PRODUTO` e `VENDA` , cada um apontando para o respectivo caminho de seu diretório de índice. Para novos índices, apenas adicione as novas opções. Veja o código:

```
public enum TipoIndice {
    PRODUTO(System.getProperty("user.home")
            + "/livro-lucene/indice-produto"),
    VENDA(System.getProperty("user.home")
            + "/livro-lucene/indice-venda");
    private String diretorio;

    TipoIndice(String diretorio) {
        this.diretorio = diretorio;
    }

    public String diretorio() {
        return diretorio;
    }
}
```

Agora, vamos para o coração de um buscador, que seria a classe central responsável pela busca e pela indexação. No nosso projeto é a classe `UtilIndice` . Ela funciona de forma similar ao buscador já visto, contudo, seu código está mais organizado e há uma novidade, que é um atributo do tipo `SearcherManager` , ou seja, este é um gerenciador de buscadores.

O `SearcherManager` foi criado para ser um dos exemplos do livro *Lucene in Action* (Michael McCandless, Erik Hatcher e Otis Gospodnetic) e está disponível desde a versão 3.5. Ele é usado para compartilhar o `IndexSearcher` entre as `Threads` do sistema de forma segura e, melhor ainda, simplificada. Acontece que ficou tão eficiente que acabou por ser utilizada largamente nos sistemas de busca.

O `SearcherManager` é um *singleton*, o que quer dizer que cada índice terá apenas uma instância deste utilitário. Isso é importante porque não pode haver duas instâncias de `IndexWriter` abertas ao mesmo tempo, sem contar que ele será compartilhado entre as requisições por aqui.

```
public class UtilIndice {
    private static Logger logger =
            Logger.getLogger(UtilIndice.class);
    private IndexWriter writer;
    private Integer quantidadeLimiteRegistros = 1000;
    private Analyzer analyzer = new BrazilianAnalyzer();
    private SearcherManager sm;
    private TipoIndice tipo;

    private static final Map<TipoIndice, UtilIndice> instancias =
            new HashMap<TipoIndice, UtilIndice>();
    // {...}
}
```

O *singleton* é garantido pelo método `getInstancia(TipoIndice)`, pelo construtor privado e pelo `abrirIndice(String)`, que estão listados no bloco a seguir. É bem simples: se aquele tipo de índice ainda não está carregado, o índice é aberto e ele é adicionado ao mapa. Caso já tenha sido aberto, é retornada sua instância. Assim:

```
    public static synchronized UtilIndice getInstancia(
            TipoIndice tipo) throws IOException {
        UtilIndice utilIndice = instancias.get(tipo);
        if (utilIndice == null) {
            utilIndice = new UtilIndice(tipo);
```

```
            instancias.put(tipo, utilIndice);
        }
        return utilIndice;
    }

    private UtilIndice(TipoIndice tipo) throws IOException {
        this.tipo = tipo;
        abrirIndice(tipo.diretorio());
        inicializaSm();
    }

    private void abrirIndice(String diretorioIndice)
            throws IOException {
        Directory diretorio =
                FSDirectory.open(Paths.get(diretorioIndice));
        analyzer = new BrazilianAnalyzer();
        IndexWriterConfig conf = new IndexWriterConfig(analyzer);
        writer = new IndexWriter(diretorio, conf);
        logger.info("IndexWriter aberto");
    }
```

O `inicializaSm()` é responsável por criar uma instância do `SearcherManager` por meio do `IndexWriter`, assim, temos um índice em tempo quase real, ou *Near Real-Time* (NRT). Quando os documentos forem adicionados ao índice com o `IndexWriter`, a busca consegue acessá-los quase imediatamente. O construtor é o `public SearcherManager(IndexWriter writer, SearcherFactory searcherFactory)`, onde o segundo parâmetro pode ser nulo e o próprio Lucene se ocupa para preencher internamente.

Há outros construtores com pequenas diferenças. Por exemplo, é possível usar o diretório do índice, sem o NRT, ou seja, as alterações do índice só serão vistas na busca depois de reabrir o buscador. Outro construtor permite aplicar as exclusões do *buffer* na busca ou forçar a gravação das exclusões direto no índice. Esta última opção é custosa e, se possível, o melhor é evitá-la.

Existem duas formas de fazer consultas: unicamente com o `IndexSearcher`, no qual o buscador abre o índice no diretório de

origem, e com o próprio `IndexWriter`, que permite a busca NRT. A partir de agora usaremos a segunda opção. Veja o novo código que usa o `IndexWriter`:

```
    private void inicializaSm() throws IOException {
        sm = new SearcherManager(writer, null);
        logger.info("SearcherManager aberto");
    }
```

Com o `UtilIndice` vamos gerenciar a criação e exclusão dos documentos no índice, em vez de usar diretamente o `IndexWriter` como antes. Nesta versão, vamos ter o `IndexWriter` e o `IndexSearcher` juntos na mesma classe. Com essa estratégia podemos usar a busca em tempo real (NRT). Assim, a classe `UtilIndice` facilita nossa vida porque esconde a API do Lucene, de forma que o programador não precisa entender os detalhes da indexação para usá-la. Quando precisar adicionar um novo item, ele executa o método `adicionarDoc`, ou quando quiser atualizar um item, apenas executa o método `atualizarDoc`.

O `UtilIndice` funciona como um ponto central para a indexação e busca. Uma classe utilitária como esta costuma ser um pouco mais complexa que as outras, no entanto, tem a vantagem de tornar a manutenção mais fácil, além de permitir otimizações de performance. A seguir estão listados os principais métodos para adicionar documentos do índice na classe `UtilIndice`. Perceba que o método apenas faz uma chamada à API do Lucene com a instrução `writer.addDocument(doc);`.

```
    public void adicionarDoc(Document doc, boolean commit)
            throws IOException {
        verificarSeAberto();
        writer.addDocument(doc);
        if (commit) {
            commit();
        }
    }

    public void adicionarDoc(Document doc) throws IOException {
```

```
        adicionarDoc(doc, true);
    }
```

Os métodos a seguir vão na mesma direção, só que fazem a atualização com a instrução `writer.updateDocument(termo, doc);` . Lembre-se de que na prática o Lucene não faz atualização; por dentro da atualização está, de fato, uma exclusão e uma nova inclusão.

```
    public void atualizarDoc(Term termo, Document doc)
            throws IOException {
        atualizarDoc(termo, doc, true);
    }

    public void atualizarDoc(Term termo, Document doc,
            boolean commit) throws IOException {
        verificarSeAberto();
        writer.updateDocument(termo, doc);
        if (commit) {
            commit();
        }
    }
```

A seguir, os métodos para remoção de documentos do índice com a instrução `writer.deleteDocuments(termo);` .

```
    public void removerDoc(String campo, String id)
            throws IOException {
        removerDoc(campo, id, true);
    }

    public void removerDoc(String campo, String id,
            boolean commit) throws IOException {
        verificarSeAberto();
        Term termo = new Term(campo, id);
        writer.deleteDocuments(termo);
        if (commit) {
            commit();
        }
    }
```

O método `UtilIndice.commit()` faz a chamada do `IndexWriter.commit()`, assim, é apenas uma delegação, mas é importante para manter o encapsulamento e a clareza do código da `UtilIndice`.

```
    public void commit() throws IOException {
        writer.commit();
    }
```

Perceba que temos o `verificarSeAberto()`, que estará presente em vários métodos da classe. Claro, ele verifica se os recursos estão abertos naquele momento porque eles podem ser fechados por uma *thread* externa. Assim, antes de usar o índice, seja para gravação ou consulta, verificamos se está tudo funcionando. Veja:

```
    private void verificarSeAberto() throws IOException {
        if (sm == null) {
            abrirIndice(tipo.diretorio());
            inicializaSm();
            logger.info("Índice reaberto");
        }
    }
```

Com o novo buscador, para executar uma busca, você recupera um `IndexSearcher` pelo `SearcherManager`, dessa forma: `IndexSearcher searcher = sm.acquire()`. Após seu uso, liberamos o recurso, ou seja, o objeto `searcher`, que volta para o *pool* do `SearcherManager`. Seria uma boa ideia não esquecer de liberar os recursos utilizados, porque eles são finitos. Para isso, use a sintaxe `sm.release(searcher)`, assim tanto a memória quanto o sistema de arquivos são liberados após seu uso.

O `sm.maybeRefresh()` é usado para retornar documentos atualizados. Além deste, temos o `sm.maybeRefreshBlocking()`, que garante instâncias sempre atualizadas, mesmo quando há várias *threads* simultâneas tentando atualizar.

De forma geral, para usar a nova classe `UtilIndice` os passos são esses: (i) abra um buscador (método `acquire`), (ii) execute a

consulta (método `search`) e (iii) libere o recurso (método `release`). Como neste bloco de exemplo:

```
public TopDocs buscar(String consulta)
        throws IOException, ParseException {
    verificarSeAberto();
    sm.maybeRefresh();
    IndexSearcher searcher = sm.acquire();
    try {
        QueryParser queryParser =
                new QueryParser("", analyzer);
        queryParser.setDefaultOperator(Operator.AND);
        Query query = queryParser.parse(consulta);
        TopDocs hits = searcher.search(query,
                quantidadeLimiteRegistros);
        return hits;
    } finally {
        sm.release(searcher);
    }
}
```

Para garantir mais flexibilidade, vamos criar mais um método `buscar`, que será usado mais para a frente neste capítulo. A diferença aqui é que a busca não é feita com uma String, e sim com um objeto `Query`, de forma que o programador pode especificar com precisão o que pretende buscar.

```
public TopDocs buscar(Query consulta)
        throws IOException, ParseException {
    verificarSeAberto();
    sm.maybeRefresh();
    IndexSearcher searcher = sm.acquire();
    try {
        TopDocs hits = searcher.search(consulta,
                quantidadeLimiteRegistros);
        return hits;
    } finally {
        sm.release(searcher);
    }
}
```

Da mesma forma, para recuperar um documento com a `UtilIndice`, use a sequência: (i) abrir (`acquire`), (ii) consultar, (iii) liberar. Veja:

```
public Document doc(int docID) throws IOException {
    verificarSeAberto();
    sm.maybeRefresh();
    IndexSearcher searcher = sm.acquire();
    try {
        Document doc = searcher.doc(docID);
        return doc;
    } finally {
        sm.release(searcher);
    }
}
```

Para finalizar, vemos a seguir o método `fechar()` que deve ser chamado após terminar suas alterações.

```
public void fechar() throws IOException {
    synchronized (this) {
        if (sm != null) {
            sm.close();
            sm = null;
            logger.info("SearcherManager fechado");
        }
        if (writer != null) {
            writer.close();
            writer = null;
            logger.info("IndexWriter fechado");
        }
    }
}
```

A questão das *threads* e da concorrência em Java não é coberta no livro, mas é bem importante para criação de sistemas distribuídos para grandes cargas, ou *big data*. Nesta seção vimos apenas superficialmente de que forma isso é implementado para garantir performance e economia de recursos. Na próxima seção veremos como usar o `UtilIndice` com os índices de exemplo.

7.3 Índice Produto

O índice `Produto` é talvez o mais importante para conseguir realizar a venda. É aqui onde o cliente realizará buscas pelos produtos de interesse. Cada documento deste índice é composto por todos os campos concatenados das tabelas `Categoria` e `Produto`. Os campos são:

Campo	Descrição
`categoriaNome`	Nome da categoria.
`produtoId`	Identificador do produto.
`produtoDescricao`	Descrição do produto.
`produtoNome`	Nome do produto.
`especFabricante`	Especificação fornecida pelo fabricante.
`produtoPreco`	Preço do produto.
`produtoPrecoPoint`	Preço do produto, usado para filtrar o resultado.
`dataAtualizacao`	Data em que o registro foi atualizado/inserido.
`textoCompleto`	Conteúdo de todos os campos concatenados.

Nesta seção vamos criar a classe que gera o índice, que é a `IndexadorProduto`, um utilitário responsável por consultar os registros no banco de dados e transformá-los em documentos do Lucene que serão indexados. As operações incluem indexar um produto, indexar todos os produtos, indexar apenas os produtos recentemente alterados e, claro, remover do índice os produtos excluídos do banco.

As operações típicas do banco de dados estão na classe `ProdutoService` e sua superclasse `GenericService`, que fazem parte do projeto no GitHub: https://github.com/masreis/e-commerce/. Nessas classes temos métodos para inclusão, alteração, exclusão e consultas às tabelas do nosso sistema. Não vamos entrar em detalhes sobre esse tema, mas é basicamente um utilitário que usa JPA. Confira na listagem o começo do nosso novo indexador:

```
public class IndexadorProduto {
    private static Logger logger =
            Logger.getLogger(IndexadorProduto.class);
    private ProdutoService produtoService = new ProdutoService();
    private Tika tika = new Tika();
    private UtilIndice utilIndice;
    // {...}
}
```

O método inicial para indexar um produto é `indexarProduto(Produto)`. Com base no objeto `Produto`, vamos pegar cada um dos seus valores e criar um objeto do tipo `Document`, que será indexado. Isto é, vamos traduzir uma entidade do banco relacional para um documento do Lucene. Aqui, a indexação será feita com o método `atualizarDoc(Term termo, Document doc, boolean commit)`, que funciona tanto para inclusão quanto para atualização de um documento. Veja o código:

```
    private void indexarProduto(Produto produto)
            throws IOException, TikaException {
        Document doc = new Document();
        StringBuilder textoCompleto = new StringBuilder();
        preencherDadosProduto(produto, doc, textoCompleto);
        preencherDadosCategoria(produto, doc, textoCompleto);
        preencherDadosTextoCompleto(doc, textoCompleto);
        utilIndice.atualizarDoc(new Term("produtoId",
                produto.getId().toString()), doc, false);
    }
```

A seguir veremos os métodos usados para preencher o documento do Lucene que será indexado. O primeiro é o `preencherDadosProduto`,

listado a seguir. Veja que o campo `produtoPreco` é um `TextField`, um tipo de campo que permite armazenar e consultar conteúdo em formato textual. Acontece que o conteúdo do `produtoPreco` é numérico e não textual.

Para campos numéricos, o Lucene tem a classe `DoublePoint` (mais detalhes no capítulo *4. Tipos de busca*, que é usada para filtrar rapidamente intervalos de valores. Por isso, criamos o `produtoPrecoPoint`. Por que precisamos dos dois? Porque o `DoublePoint` não armazena o conteúdo do campo, ele só permite a consulta. Vamos usar a eficiência na consulta do `DoublePoint` e o valor armazenado no campo do tipo `TextField`.

No final do método temos o preenchimento do `textoCompleto` com os dados textuais do produto. Este campo será usado para pesquisa livre e acredito que não seja interessante aqui colocar os valores numéricos como preço ou identificador do produto. Imagine que, se o usuário quiser procurar por produtos em uma faixa de preço, ele o fará no campo `produtoPrecoPoint`, como será visto mais tarde neste capítulo. Da mesma forma, se quiser procurar por um produto específico, tem de pesquisar pelo campo `produtoId`.

```
    private void preencherDadosProduto(Produto produto,
            Document doc, StringBuilder textoCompleto)
            throws IOException, TikaException {
        doc.add(new StringField("produtoId",
                produto.getId().toString(), Store.YES));
        String descricao = produto.getDescricao() == null ? ""
                : produto.getDescricao();
        doc.add(new TextField("produtoDescricao", descricao,
                Store.YES));
        doc.add(new TextField("produtoNome", produto.getNome(),
                Store.YES));
        String especFabricante = getEspecProduto(produto);
        if (!"".equals(especFabricante)) {
            doc.add(new TextField("especFabricante",
                    especFabricante, Store.YES));
        }
        doc.add(new TextField("produtoPreco",
```

```
                produto.getPreco().toString(), Store.YES));
        doc.add(new DoublePoint("produtoPrecoPoint",
                produto.getPreco().doubleValue()));
        doc.add(new TextField("dataAtualizacao",
                DateTools.dateToString(
                        produto.getDataAtualizacao(),
                        Resolution.MINUTE),
                Store.YES));
        textoCompleto.append(" ");
        textoCompleto.append(produto.getNome());
        if (!"".equals(especFabricante)) {
            textoCompleto.append(" ");
            textoCompleto.append(especFabricante);
        }
        textoCompleto.append(" ");
        textoCompleto.append(produto.getDescricao());
    }
```

O método `getEspecProduto(Produto)` usa o Apache Tika para extrair o conteúdo da especificação do fabricante, caso ela exista em formato PDF, DOC etc. O Tika foi detalhado no capítulo *3. Indexação e busca*.

```
    public String getEspecProduto(Produto produto)
            throws IOException, TikaException {
        if (produto.getEspecificacaoFabricante() != null) {
            ByteArrayInputStream bytes =
                    new ByteArrayInputStream(produto
                            .getEspecificacaoFabricante());
            return tika.parseToString(bytes);
        }
        return "";
    }
```

Não há muitas novidades aqui, apenas a criação de campos do Lucene, um `StringField` para o `produtoId` e os demais são `TextField`. Lembre-se de que os campos de identificação como CPF ou ID são `StringField`, enquanto os campos de texto comum são `TextField`. Vamos ao `preencherDadosCategoria`. Veja:

```
private void preencherDadosCategoria(Produto produto,
        Document doc, StringBuilder textoCompleto) {
    for (Categoria categoria : produto.getCategorias()) {
        doc.add(new TextField("categoriaNome",
                categoria.getNome(), Store.YES));
        textoCompleto.append(" ");
        textoCompleto.append(categoria.getNome());
    }
}
```

Perceba que um produto pode ter várias categorias e todas estarão associadas a um único documento. É um campo multivalorado, ou seja, um único campo pode ter diversos valores. Neste caso, `categoriaNome` pode aparecer mais de uma vez para o mesmo produto, porque o mesmo item pode estar de fato em várias categorias.

Outra possibilidade para trabalhar com campos multivalorados é fazer como no campo `textoCompleto`, que concatena o valor dos demais campos. Em vez de ter vários campos `categoriaNome`, você pode ter apenas um com o nome de todas as categorias concatenados. É possível usar ambas no mesmo projeto, sem problemas.

Exemplo: considere o produto "Celular Motorola G4", que pertence às categorias `Telefonia`, `Utilidades domésticas` e `Eletrônicos`.

Abordagem 1 (com campo multivalorado): o campo `categoriaNome` é repetido com valores diferentes.

Campo	Descrição
`produtoNome`	Celular Motorola G4.
`categoriaNome`	Telefonia.
`categoriaNome`	Eletrônicos.
`categoriaNome`	Utilidades domésticas.

Abordagem 2 (campo único): o campo `categoriaNome` não é repetido e tem o mesmo conteúdo da abordagem 1.

Campo	Descrição
`produtoNome`	Celular Motorola G4.
`categoriaNome`	Telefonia Utilidades domésticas Eletrônicos.

Veja que as duas soluções resolvem o problema: dada uma categoria, o cliente vai encontrar o produto desejado. A abordagem 1 permite utilizar recursos mais complexos como *facets* e buscas mais específicas. A abordagem 2 (campo único) tem a vantagem de ser bastante prática para busca simples por palavra-chave.

O último bloco analisado será o `preencherDadosTextoCompleto`. Neste caso nós usamos o campo `textoCompleto`, onde concatenamos os valores textuais dos campos das classes, separamos com espaços em branco entre cada palavra. Como sugerido, é uma consulta livre por qualquer palavra do documento, ou seja, vai recuperar os produtos que contenham o termo pesquisado em qualquer campo textual.

Por exemplo: em vez de consultar os produtos que têm *xbox* no campo `nomeProduto`, na busca livre o Lucene vai procurar os produtos que contém *xbox* em qualquer campo, não apenas no campo `nomeProduto`. Veja o código:

```
    private void preencherDadosTextoCompleto(Document doc,
            StringBuilder textoCompleto) {
        doc.add(new TextField("textoCompleto",
                textoCompleto.toString(), Store.YES));
    }
```

A classe tem um método para indexar todos os produtos. Ele recupera uma lista com esses itens e chama o método `indexarProduto(Produto)` para cada um. Será usado na próxima seção para criar o índice inicial do sistema.

```
public void indexarProdutos()
        throws IOException, TikaException {
    List<Produto> produtos =
            produtoService.carregarColecao(Produto.class);
    for (Produto prod : produtos) {
        indexarProduto(prod);
    }
}
```

Depois de criar o índice, precisamos acompanhar as modificações e fazer as atualizações. Quando muda o preço de um item, o índice tem de refletir aquela modificação. Da mesma forma, quando um novo produto é adicionado, ele deve fazer parte do índice e, evidentemente, quando o produto é excluído deve também ser removido do índice. Para resolver essas demandas, temos o método `atualizarIndice(int tempoEmMinutos)`. Ele verifica quais produtos foram alterados e faz a reindexação. Caso o produto esteja inativo, ele remove do índice.

Quando um registro é alterado, o campo `dataAtualizacao` é atualizado com a data e hora atual. Com isso, o método `ProdutoService.consultarAtualizacoes(int)` recupera apenas aqueles itens que foram alterados recentemente.

Outro campo importante é o `ativo`, que indica se aquele registro é válido ou deve ser desconsiderado, é o que chamamos de exclusão lógica, isto é, o registro não foi excluído da tabela, apenas está marcado como inativo. Veja o código:

```
public void atualizarIndice(int tempoEmMinutos)
        throws IOException, TikaException {
    List<Produto> produtos = produtoService
            .consultarAtualizacoes(tempoEmMinutos);
    logger.info(produtos.size() + " produtos alterados");
    for (Produto produto : produtos) {
        if (produto.isAtivo()) {
            indexarProduto(produto);
        } else {
            removerProdutoIndice(produto);
```

```
            }
        }
        logger.info("Indexação concluída");
    }
```

Para finalizar o `IndexadorProduto`, a seguir estão os métodos para inicializar, remover e fechar os recursos. Perceba que o `UtilIndice` está apontando para o tipo de índice `Produto`, mas pode ser reutilizado em outros índices que, eventualmente, sejam criados no projeto.

```
    public void inicializar() throws IOException {
        utilIndice = UtilIndice.getInstancia(TipoIndice.PRODUTO);
    }

    private void removerProdutoIndice(Produto produto)
            throws IOException {
        utilIndice.removerDoc("produtoId",
                produto.getId().toString());
    }

    public void fechar() throws IOException {
        utilIndice.fechar();
    }
```

O `IndexadorProduto` faz o trabalho de indexação e busca referente ao índice de produto. A partir dele, vamos criar o `IndexadorVenda` para o índice de vendas, ou para qualquer outro. Na próxima seção vamos carregar uma base de dados de exemplo para validar os utilitários.

7.4 Carga inicial dos dados

Nosso projeto de e-commerce conta com uma base de dados que pode ser usada nos testes. Os registros estão gravados no arquivo `dump.sql` que está disponível no GitHub (https://github.com/masreis/e-commerce). Nesta seção explicaremos como fazer sua importação.

Comece criando a base de dados no MySQL com o comando `mysql -u root -p -e "create database ecommerce"` . Depois, execute o comando de importação `mysql -u root -p ecommerce < dump.sql` . Apenas se certifique de indicar o caminho correto do arquivo `dump.sql` , que deve estar no diretório `src/test/resources/` , dentro da pasta do projeto `e-commerce` . Com isso, as tabelas serão criadas e populadas com valores fictícios. O parâmetro `-p` do MySQL é opcional, usado para solicitar a senha para o usuário.

Foram criadas classes de teste para carregar algumas tabelas, como a de vendas e de clientes. As classes são `CriarVendasTest` , `CriarClientesTeste` e `AlterarPrecoProdutoTest` . Veja que os nomes dos clientes foram criados com caracteres aleatórios, bem como os preços dos produtos, que não representam o valor real de mercado. A classe `AlterarPrecoProdutoTest` muda os preços dos produtos e pode ser usada para simular um ambiente real, onde os valores são de fato alterados durante a operação.

Na sequência, temos de criar o índice no Lucene para esta base de dados inicial. A classe que indexa os dados do MySQL é `IndexadorProdutoTest` . Para executá-la no Eclipse, use a opção *Run as JUnit Test*. Aqui, ela vai apenas chamar os métodos já vistos anteriormente (`iniciarlizar` , `indexarProdutos` e `fechar`).

```
public class IndexadorProdutoTest {
    @Test
    public void testIndexarTodosProdutos()
            throws IOException, TikaException {
        IndexadorProduto indexador = new IndexadorProduto();
        indexador.inicializar();
        indexador.indexarProdutos();
        indexador.fechar();
    }
}
```

Ao final dessa execução, teremos o índice gerado e o usuário poderá consultar os produtos de teste cadastrados na página de busca da aplicação web. As configurações de acesso ao MySQL

estão no arquivo `/src/test/resources/META-INF/persistence.xml` . Inicie o Tomcat e acesse a URL do sistema (http://localhost:8080/e-commerce/publico/busca.faces). A imagem a seguir mostra a tela de busca livre do sistema:

Figura 7.2: Tela de busca livre.

Vamos fazer uma consulta simples, como *xbox*. Veja na imagem o resultado da busca:

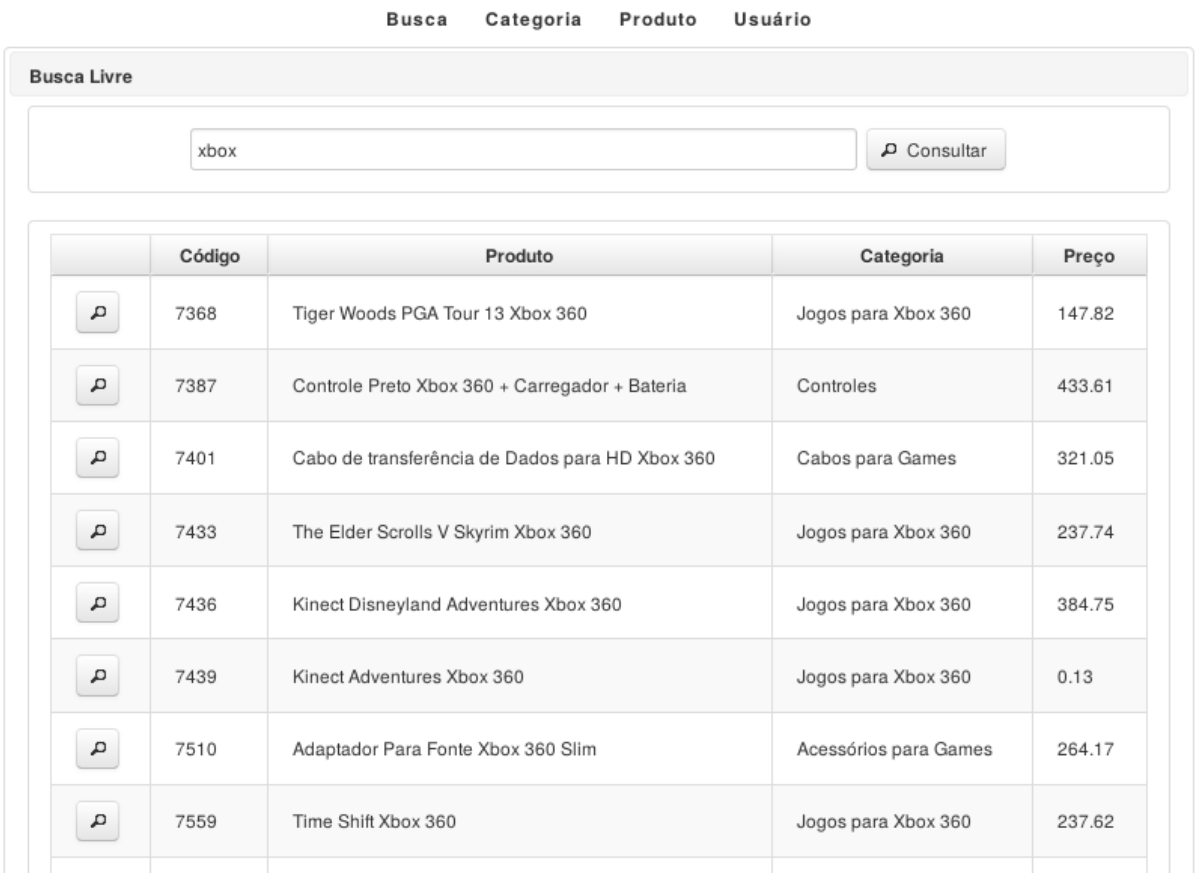


Figura 7.3: Busca por *xbox*.

Clique na lupa para visualizar uma pequena descrição do produto. É importante notar que esse texto foi carregado na página web por meio do Lucene e não do banco de dados.

Figura 7.4: Descrição do produto.

Com esse protótipo, temos o nosso próprio buscador web, que pode ser adaptado para outros domínios. Nas próximas seções, discutiremos os detalhes de implementação das páginas JSF. Se não conhece JSF, ou se as suas telas serão implementadas com outra tecnologia, pode simplesmente pular a próxima seção.

7.5 O managed bean BuscaLivreProdutoBean

Estamos trabalhando com JSF, até porque este é o mecanismo para criação de interfaces web do Java EE, então, nada mais natural do que utilizá-lo para construir nossas telas. A tela em questão é a de busca livre, ou seja, aquela onde o usuário vai digitar um termo qualquer e o sistema deverá encontrar os produtos correspondentes.

A classe que gerencia a tela de busca é a `BuscaLivreProdutoBean`. O objetivo dela é executar as buscas, coletar o resultado do índice no Lucene e mostrar os itens formatados na página web. A seguir está uma parte do código:

```
@ManagedBean
@ViewScoped
public class BuscaLivreProdutoBean extends BaseBean {
    private static final long serialVersionUID =
            -7508553590263034662L;
    private String consulta;
    private LuceneLazyDataModel docs;
    private TipoIndice tipo = TipoIndice.PRODUTO;
    // {...}
}
```

O método mais importante aqui é o `consultar()`, que está logo a seguir. Este método será executado quando o usuário digitar o nome do produto que pretende buscar e clicar no botão "Consultar". Em seguida, o sistema vai consultar o índice e retornar uma coleção de itens, armazenados em um objeto do tipo `LuceneLazyDataModel`. Esta classe é detalhada na próxima seção. Os parâmetros do seu construtor recebem a consulta do usuário e o tipo de índice, que neste caso é `Produto`, mas também pode ser `Venda`, que será visto mais adiante neste capítulo.

```
    public void consultar() {
        docs = new LuceneLazyDataModel(getConsulta(), getTipo());
    }
```

Quando o usuário navegar pelas páginas do resultado da busca, o *managed bean* vai carregar os registros no método `getDocs()`, que está logo a seguir. Perceba que cada vez que o usuário mudar de página é executada uma nova consulta no Lucene, criando um novo objeto `LuceneLazyDataModel`.

```
    public LuceneLazyDataModel getDocs() {
        if (docs == null) {
            docs = new LuceneLazyDataModel(getConsulta(),
                    getTipo());
        }
        return docs;
    }

    public void setDocs(LuceneLazyDataModel docs) {
```

```
        this.docs = docs;
    }
```

O `getDuracaoBusca` mostra o tempo de duração de uma busca com nosso sistema.

```
    public BigDecimal getDuracaoBusca() {
        Double d = getDocs().getDuracaoBusca() / 1000d;
        BigDecimal bd = new BigDecimal(d).setScale(4,
                BigDecimal.ROUND_CEILING);
        return bd;
    }
```

O `getDescricaoFormatada` é um método para converter a quebra de linha do texto (`\n`) em quebra de linha no HTML (`<br />`).

```
    public String getDescricaoFormatada() {
        return getDocs().getRowData().get("produtoDescricao")
                .replaceAll("\n", "<br />");
    }
```

Os últimos itens são o `setConsulta` e o `getConsulta` , que são os métodos de acesso para recuperar o texto pesquisado pelo usuário.

```
    public void setConsulta(String consulta) {
        this.consulta = consulta;
    }

    public String getConsulta() {
        return consulta;
    }
```

Uma busca por texto pode retornar uma grande quantidade de registros, sendo necessário que o resultado seja particionado em diversas páginas. Esta é a função do `LuceneLazyDataModel` . Ele faz a paginação do resultado da busca.

7.6 Paginação com Lucene

Uma busca pode retornar apenas um item, da mesma forma que pode retornar um milhão de itens. E convém notar que mostrar mil registros em uma página web não é recomendado, porque o usuário não lê toda essa quantidade. Por isso, vamos usar a paginação e mostrar apenas poucos itens em cada página.

A paginação é uma técnica de otimização utilizada na visualização de grandes conjuntos de dados. No nosso projeto será implementada com um recurso específico do PrimeFaces, mas está disponível em outras ferramentas de *front-end*. É sobretudo importante em sistemas na internet, porque uma página web mais extensa é um grande problema de performance, mesmo em máquinas modernas.

Quando usamos paginação, mesmo que o resultado da consulta tenha milhares de itens, o sistema carrega apenas aqueles mostrados na página aberta pelo usuário. Isso economiza memória e processamento no servidor de aplicação e no computador do usuário porque o número de registros carregados é limitado.

Exemplo: um resultado com 105 itens e paginação com 20 itens significa que serão 5 páginas com 20 itens e uma página com os 5 elementos finais. Quando o usuário estiver navegando pela página 1, o sistema carrega apenas 20 itens na página. Nem precisa dizer que é muito mais rápido que carregar tudo de uma vez. A imagem a seguir mostra como fica a tela paginada:

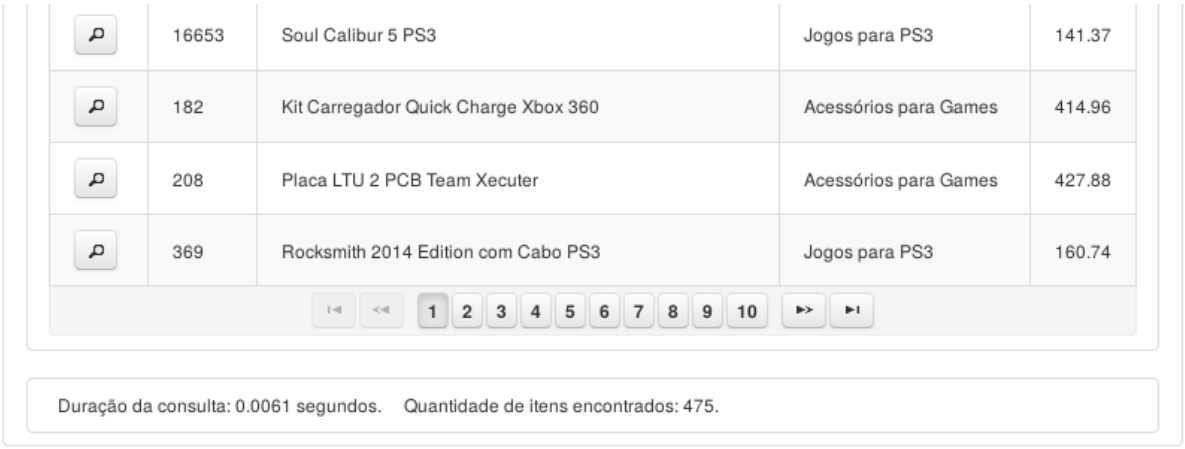


Figura 7.5: Paginação no PrimeFaces.

No PrimeFaces, a paginação usa como base a classe abstrata `org.primefaces.model.LazyDataModel`. Nesta classe, o método `load(int first, int pageSize, String sortField, SortOrder sortOrder, Map<String,Object> filters)` é responsável por carregar os registros de cada página. Este método é executado pelo próprio PrimeFaces quando o usuário muda de página. Para a paginação funcionar com o Lucene, foi criada a subclasse `LuceneLazyDataModel`, bem como uma implementação específica do método `load`.

Esta é a parte inicial da nossa classe, com os atributos `consulta`, `duracaoBusca` e `tipo`. A `consulta` é o texto pesquisado pelo usuário, a `duracaoBusca` armazena o tempo necessário para retornar o resultado e o `tipo` informa se é um índice de produto ou de venda.

```
public class LuceneLazyDataModel
        extends LazyDataModel<Document> {
    private static final long serialVersionUID =
            1153244287993412470L;
    private String consulta;
    private long duracaoBusca;
    private TipoIndice tipo;

    public LuceneLazyDataModel(String consulta,
            TipoIndice tipo) {
        this.consulta = consulta;
        this.tipo = tipo;
```

```
    }
    // {...}
}
```

E agora, o mais importante, o `load`, método que carrega os dados na tela. O parâmetro `first` indica qual o primeiro item do *array* de resultado deve ser mostrado. O `pageSize` indica quantos itens, a partir do primeiro, serão mostrados. Só isso já é suficiente para montar a paginação. O `sortOrder` configura a ordenação, enquanto o `filters` é um mapa usado para eventuais filtros.

Cada vez que o usuário muda de página, incluindo a primeira carga, o método `load` é executado. Como nossa implementação é específica para o Lucene, temos o `UtilIndice` para realizar as buscas no campo `textoCompleto`. O atributo `duracaoBusca` mostra em milissegundos o tempo necessário para cada mudança de página.

```
    public List<Document> load(int first, int pageSize,
            String sortField, SortOrder sortOrder,
            Map<String, Object> filters) {
        long inicio = System.currentTimeMillis();
        // Nova consulta cada vez que o usuário muda de página
        try {
            UtilIndice utilIndice =
                    UtilIndice.getInstancia(tipo);
            String consultaTextoCompleto =
                    "textoCompleto:(" + consulta + ")";
            TopDocs hits =
                    utilIndice.buscar(consultaTextoCompleto);
            List<Document> lista = new ArrayList<Document>();
            // Carrega apenas os itens daquela página
            for (int i = first; i < first + pageSize; i++) {
                if (i >= hits.totalHits) {
                    break;
                }
                int idDoc = hits.scoreDocs[i].doc;
                Document documento = utilIndice.doc(idDoc);
                lista.add(documento);
            }
            // Informa a quantidade total de itens da consulta
```

```
            setRowCount(hits.totalHits);
            duracaoBusca = System.currentTimeMillis() - inicio;
            return lista;
        } catch (Exception e) {
            throw new RuntimeException(e);
        }
    }
```

O método `load` faz o trabalho difícil de consultar e mostrar o resultado paginado em uma página web. Dando prosseguimento ao nosso trabalho, a seguir veremos a página de busca escrita em `xhtml`.

A página busca.xhtml

A página `busca.xhtml` mostra o resultado da busca com o recurso da paginação, que pôde ser conferido na figura *Paginação no PrimeFaces*. A seguir está listado um trecho de código desta página. O código completo pode está disponível no repositório do projeto no GitHub (https://github.com/masreis/e-commerce/).

Você pode ver que temos um componente visual do PrimeFaces, o `p:dataTable`, onde os valores serão preenchidos pelo atributo `docs` do *managed bean* `BuscaLivreProdutoBean`. A opção `paginator="true"` permite a paginação do resultado. São várias colunas (`p:column`) não mostradas aqui, cada uma representando um campo indexado do produto.

Estamos usando recursos específicos do PrimeFaces e do Java EE, como no caso da anotação `@ViewScoped`, que permite a utilização de AJAX na página web, assim, não seria um grande problema migrar para outros produtos, mesmo fora do JSF, como AngularJS ou jQuery. Aqui está uma parte da página `busca.xhtml`:

```
<p:dataTable id="tblDocs" value="#{buscaLivreProdutoBean.docs}"
    tableStyle="table-layout:auto;width:100%;"
    rendered="#{not empty buscaLivreProdutoBean.docs}"
    widgetVar="tblDocsWidget" paginator="true"
    paginatorPosition="bottom" var="doc" rows="20"
```

```
    paginatorAlwaysVisible="false" lazy="true">
    <p:ajax event="page" update=":formPesquisa:painelStatus" />

    {...}

    <p:column>
        <f:facet name="header">
            <p:outputLabel value="Preço" />
        </f:facet>
        <p:outputLabel escape="false"
                value="#{doc.get('produtoPreco')}" />
    </p:column>

</p:dataTable>
```

Neste trecho de código conseguimos ver o código necessário para utilizar um recurso avançado do PrimeFaces, neste caso, a paginação com Lucene. Deve-se notar que a página JSF é usada para apresentação visual do resultado, mas é possível sua utilização em qualquer outra tecnologia de *front-end* com uma pequena adaptação. Na sequência veremos como o índice se comporta durante a atualização, reproduzindo um sistema real em produção.

7.7 Atualizando o índice com temporizador

Os sistemas web costumam ter altas taxas de atualização, ou seja, há muita modificação nas tabelas com a inclusão, alteração ou exclusão de registros. Os bancos relacionais foram projetados para este tipo de situação, em que temos muitas transações e confirmações (*commits*). Com isso, uma alteração na base fica imediatamente disponível assim que é confirmada.

No caso do Lucene, ao contrário, não é indicado o uso do *commit* para cada alteração, porque o I/O (leitura e escrita) de disco pode (e certamente vai) se transformar em um gargalo de performance. O mais interessante é o uso de atualizações em lote (*batch*). O sistema atualiza vários documentos no índice de uma vez, em intervalos de tempo predefinidos. O importante é que não devemos fazer *commit* a cada operação de inclusão, alteração ou exclusão. No capítulo *5. Principais classes do Lucene* há mais detalhes e experimentos com a indexação e a configuração dos parâmetros de memória. A atualização em *batch* junto com o NRT oferece praticamente a mesma taxa de atualização de um banco relacional, com a vantagem de ter as buscas mais eficientes e com hardware mais barato.

Para tratar dessa questão da indexação em lote criamos a classe `TemporizadoIndiceProduto`. Ela indexa no Lucene os registros do banco de dados que foram atualizados em um determinado período de tempo. Assim, quando um produto é incluído, alterado ou excluído, o campo `dataAtualizacao` recebe a data e hora atual.

O `TemporizadorIndiceProduto` implementa `Runnable`, ou seja, é uma `Thread` e tem um agendador para executar a atualização do índice a cada minuto com o `IndexadorProduto.atualizarIndice(int)`. O código a seguir mostra como isso é feito:

```
public class TemporizadorIndiceProduto implements Runnable {
    private static Logger logger =
            Logger.getLogger(TemporizadorIndiceProduto.class);

    private final ScheduledExecutorService scheduler =
            Executors.newScheduledThreadPool(1);

    public void iniciar() {
        scheduler.scheduleAtFixedRate(this, 1, 1,
                TimeUnit.MINUTES);
    }

    public void run() {
```

```
        IndexadorProduto indexador = new IndexadorProduto();
        try {
            logger.info("Iniciando temporizador "
                    + Thread.currentThread().getName());
            indexador.inicializar();
            int um_minuto = 1;
            indexador.atualizarIndice(um_minuto);
            indexador.fechar();
            logger.info("Finalizando temporizador "
                    + Thread.currentThread().getName());
        } catch (Exception e) {
            logger.error(e);
        }
    }
}
```

Para ver o mecanismo funcionando, o primeiro passo é subir o servidor de aplicação. Estamos usando o Tomcat, que é uma boa alternativa por ser simples e grátis. Depois, é preciso alterar alguns registros, caso contrário o indexador não terá motivo para atualizar o índice. É o que faz a classe `AlterarPrecoProdutoTest`. Ela executa a alteração de diversos produtos de forma aleatória, simulando a operação de mudança de preços tradicional. É uma facilidade para que você não tenha que ficar alterando manualmente, o que daria muito mais trabalho.

Ela consulta todos os produtos na tabela e altera o seu preço com um valor aleatório e a `dataAtualizacao` com a data e hora atual, assim, o registro fica marcado para ser reindexado na próxima execução do `TemporizadorIndiceProduto`. Veja o código:

```
public class AlterarPrecoProdutoTest {
    private EntityManager em;
    private List<Produto> produtos;

    @Before
    public void inicializar() {
        em = JPAUtil.getInstance().getEntityManager();
        produtos = em.createQuery("select p from Produto p",
                Produto.class).getResultList();
```

```
    }

    @After
    public void finalizar() {
        em.close();
    }

    @Test
    public void testAlterarPrecos() throws InterruptedException {
        for (Produto p : produtos) {
            em.getTransaction().begin();
            double preco =
                    ThreadLocalRandom.current().nextDouble(500);
            p.setPreco(BigDecimal.valueOf(preco));
            p.setDataAtualizacao(new Date());
            em.persist(p);
            em.getTransaction().commit();
        }
    }
}
```

7.8 Índice Venda

Vamos criar agora o índice `Venda`. A lógica é parecida com a do índice de `Produto`, ou seja, em vez de pesquisar na tabela `Venda` do banco relacional, agora vamos pesquisar no índice `Venda` do Lucene. A lista de campos é diferente do índice de `Produto` porque o objetivo aqui é encontrar o histórico do que o cliente comprou no site. Estamos usando os valores de todos os campos envolvidos em uma venda para gerar este índice, o que traz flexibilidade para a aplicação, além de dispensar o uso do banco de dados.

Outra observação importante é que o registro de uma venda contém os identificadores (`produtoId`, `clienteId` etc.) e suas descrições (`produtoNome`, `produtoDescricao`, `clienteNome` etc.). É uma estratégia que traz flexibilidade para a busca. O usuário pode descobrir, por

exemplo, as compras de um cliente específico, as vendas de um produto, enfim, é possível consultar por qualquer um dos campos utilizados no índice.

A lista de campos utilizada neste índice está logo a seguir. Perceba que é junção dos campos das tabelas `Cliente`, `ItemVenda`, `Produto` e `Venda`. Os campos são multivalorados, então, caso um produto tenha mais de um item, teremos vários campos `itemProdutoNome` com valores diferentes associados à mesma venda.

Campo	Descrição
`vendaId`	Identificação da venda.
`data`	Data da venda.
`dataAtualizacao`	Data de atualização do registro da venda.
`itemQuantidade`	Quantidade de produtos de um item de venda.
`itemValorUnitario`	Valor unitário do produto no item de venda.
`itemValorTotal`	Valor total do produto no item de venda.
`itemValorUnitarioPoint`	Valor unitário do produto no item de venda (para filtrar o resultado).
`itemValorTotalPoint`	Valor total do produto no item de venda, usado para filtrar.
`itemProdutoId`	Identificador do produto do item de venda.
`itemProdutoDescricao`	Descrição do produto do item de venda.
`itemProdutoNome`	Nome do produto do item de venda.

Campo	Descrição
`itemProdutoEspecFabricante`	Especificação fornecida pelo fabricante.
`itemProdutoPreco`	Preço do produto no item de venda.
`itemProdutoPrecoPoint`	Preço do produto no item de venda (filtro).
`clienteId`	Identificação do cliente.
`clienteNome`	Nome do cliente.
`clienteEmail`	E-mail do cliente.
`textoCompleto`	Conteúdo de todos os campos concatenados.

Para indexar as vendas, foi criada a classe `IndexadorVenda`, que funciona da mesma forma que `IndexadorProduto`. Agora, durante a inicialização, use o `TipoIndice.VENDA` para recuperar a instância de `UtilIndice`, reutilizando o utilitário do índice.

```
public void inicializar() throws IOException {
    utilIndice = UtilIndice.getInstancia(TipoIndice.VENDA);
}
```

O método para indexar uma venda considera a lista de campos vista nesta seção. Está dividido em quatro outros métodos para processar separadamente os dados da tabela `Venda`, `ItemVenda`, `Cliente` e mais uma vez aparece a ideia do campo `textoCompleto`, com todos os outros campos concatenados. Perceba que não temos a categoria do produto neste índice, assim, não será possível pesquisar por esse campo.

```
private void indexarVenda(Venda venda)
        throws IOException, TikaException {
    Document doc = new Document();
    StringBuilder textoCompleto = new StringBuilder();
    preencherDadosVenda(venda, doc);
```

```
        preencherDadosItemVenda(venda, doc, textoCompleto);
        preencherDadosCliente(venda, doc, textoCompleto);
        preencherDadosTextoCompleto(venda, doc, textoCompleto);
        utilIndice.atualizarDoc(
                new Term("vendaId", venda.getId().toString()),
                doc, false);
    }
```

Os métodos para preencher os campos recuperam cada um dos valores do objeto `Venda` e fazem o mapeamento para um objeto do tipo `Document`. A estratégia é igual à adotada nos exemplos anteriores, mas vamos lá.

Em `preencherDadosVenda`, pegamos os campos da venda (`vendaId`, `dataAtualizacao` e `data`) e os atribuímos ao objeto `Document`. Não estamos preenchendo o atributo `textoCompleto`, mas se achar necessário, use a estratégia adotada no índice de produto.

```
    private void preencherDadosVenda(Venda venda, Document doc,
            StringBuilder textoCompleto) {
        doc.add(new StringField("vendaId",
                venda.getId().toString(), Store.YES));
        doc.add(new TextField("dataAtualizacao",
                DateTools.dateToString(
                        venda.getDataAtualizacao(),
                        Resolution.MINUTE),
                Store.YES));
        doc.add(new TextField("data", DateTools.dateToString(
                venda.getData(), Resolution.MINUTE), Store.YES));
    }
```

Em `preencherDadosCliente`, os dados do cliente que efetivou a venda são indexados e também preenchemos o campo `textoCompleto`. Contudo, uma busca livre com o nome do cliente pode ser pouco precisa, ou seja, buscar todas as vendas onde aparece `marco` retornará muitos registros, assim, talvez seja melhor fazer um refinamento. Dá para ver que a modelagem dos campos que serão indexados não é mesmo uma ciência exata.

```
private void preencherDadosCliente(Venda venda, Document doc,
        StringBuilder textoCompleto) {
    doc.add(new StringField("clienteId",
            venda.getCliente().getId().toString(),
            Store.YES));
    doc.add(new TextField("clienteNome",
            venda.getCliente().getNome(), Store.YES));
    doc.add(new StringField("clienteEmail",
            venda.getCliente().getEmail(), Store.YES));
    textoCompleto.append(" ");
    textoCompleto.append(venda.getCliente().getNome());
    textoCompleto.append(" ");
    textoCompleto.append(venda.getCliente().getEmail());
}
```

Uma venda contém ao menos um `ItemVenda`, e o método `preencherDadosItemVenda` é responsável por esse processamento. Note que ao final temos uma chamada para `preencherDadosProduto`, porque cada item deve se referir a um produto, então, quando indexar um `ItemVenda`, deve-se também indexar os dados do produto associado.

```
private void preencherDadosItemVenda(Venda venda,
        Document doc, StringBuilder textoCompleto)
        throws IOException, TikaException {
    for (ItemVenda iv : venda.getItensVenda()) {
        doc.add(new TextField("itemQuantidade",
                iv.getQuantidade().toString(), Store.YES));
        doc.add(new TextField("itemValorUnitario",
                iv.getValorUnitario().toString(),
                Store.YES));
        doc.add(new TextField("itemValorTotal",
                iv.getValorTotal().toString(), Store.YES));
        doc.add(new IntPoint("itemQuantidadePoint",
                iv.getQuantidade()));
        doc.add(new DoublePoint("itemValorUnitarioPoint",
                iv.getValorUnitario().doubleValue()));
        doc.add(new DoublePoint("itemValorTotalPoint",
                iv.getValorTotal().doubleValue()));
        preencherDadosProduto(iv.getProduto(), doc,
                textoCompleto);
```

```
    }
}
```

Os métodos `preencherDadosProduto` e `preencherDadosTextoCompleto` são iguais aos vistos anteriormente e não precisamos repeti-los aqui.

Para criar dados de teste, temos a classe `CriarVendasTest`. Ela pode ser usada para criar registros de vendas. Para definir a quantidade de simulações, veja a variável `QUANTIDADE_VENDAS` que, no GitHub, está definida como `100 * 1000`, ou cem mil vendas. Dependendo do poder de processamento da máquina, essa operação pode demorar algumas horas. O principal método é o `testCriarVendas`, listado aqui. Veja que a ideia é criar vendas aleatórias.

```
public void testCriarVendas() {
    // Quantidade de vendas de teste
    for (int i = 0; i < QUANTIDADE_VENDAS; i++) {
        em.getTransaction().begin();
        Venda venda = new Venda();
        // Cada venda tem até 10 itens aleatórios
        int qtdItensVenda = ThreadLocalRandom.current()
                .nextInt(1, 10 + 1);
        venda.setCliente(getClienteAleatorio());
        venda.setData(getDataAleatoria());
        venda.setDataAtualizacao(new Date());
        venda.setAtivo(true);
        for (int j = 0; j < qtdItensVenda; j++) {
            ItemVenda itemVenda = new ItemVenda();
            Produto produto = getProdutoAleatorio();
            itemVenda.setProduto(produto);
            // Cada item pode ter até 10 unidades na venda
            int qtdItem =
                    ThreadLocalRandom.current().nextInt(10);
            itemVenda.setQuantidade(qtdItem);
            double total = produto.getPreco().doubleValue()
                    * qtdItem;
            itemVenda.setValorTotal(
                    BigDecimal.valueOf(total));
            itemVenda.setValorUnitario(produto.getPreco());
            venda.getItensVenda().add(itemVenda);
```

```
            }
            em.persist(venda);
            em.getTransaction().commit();
        }
    }
```

O próximo passo é indexar as vendas, o que faremos com o `IndexadorVendaTest`. Ele é semelhante ao `IndexadorProdutoTest`, como você pode imaginar. Após criar algumas vendas simuladas, execute o teste para criar o novo índice de `Venda`.

```
public class IndexadorVendaTest {
    @Test
    public void testIndexarTodasVendas()
            throws IOException, TikaException {
        IndexadorVenda indexador = new IndexadorVenda();
        indexador.inicializar();
        indexador.indexarVendas();
        indexador.fechar();
    }
}
```

Uma vez que o índice de `Venda` está pronto, podemos subir o servidor de aplicação, no nosso caso é o Tomcat, que ativará o `TemporizadorIndiceVenda` por meio de um `WebListener`, visto no bloco de código adiante.

```
@WebListener
public class ConfiguracaoTemporizador
        implements ServletContextListener {

    public void contextInitialized(ServletContextEvent event) {
        new TemporizadorIndiceProduto().iniciar();
        new TemporizadorIndiceVenda().iniciar();
    }

    public void contextDestroyed(ServletContextEvent event) {
    }

}
```

O índice de vendas é similar ao índice de produto. A novidade aqui fica por conta do temporizador, um recurso usado para fazer otimizar o uso dos recursos evitando grandes operações de I/O no disco do servidor.

Resumo

O projeto desenvolvido ao longo do capítulo mostrou uma estratégia para incorporar um motor de buscas a um sistema preexistente, no nosso caso uma solução de *e-commerce*. Foram apresentados recursos avançados como a paginação, o `SearcherManager`, suporte a múltiplos índices, a busca em NRT e a indexação com temporizadores. Estas ideias podem ser reutilizadas em outros sistemas ou até mesmo em outras tecnologias de *front-end*, para quem não usa JSF.

Para sistemas que usam JPA e Hibernate há ainda outra opção, que seria o Hibernate Search, visto no próximo capítulo. O Hibernate Search permite aproveitar alguns recursos do JPA, como as classes de entidade e o `EntityManager`, com as vantagens do Lucene.

Capítulo 8
Hibernate Search ORM

O Hibernate Search ORM é um subprojeto do Hibernate que combina a funcionalidade de busca textual com as facilidades do JPA. Seu uso é possível em sistemas que já contam com o JPA e o Hibernate, pois estes são pré-requisitos para usar o Hibernate Search. Com ele, cada classe de entidade (`@Entity`) do JPA se torna um índice independente do Lucene, porque o Hibernate Search usa internamente o próprio Lucene. Assim, não é necessário pensar na modelagem do índice, porque cada tabela será transformada automaticamente em um índice do Lucene.

Com o uso de poucas anotações, é possível criar o índice, enquanto o próprio Hibernate Search cuida da atualização, mantendo a sincronização com os dados do banco. Com ele, temos o poder de consulta do Lucene e a facilidade de mapeamento do Hibernate. A partir da versão 5.6, o Hibernate Search pode acessar o Elasticsearch como alternativa ao Lucene.

Para mostrar seu funcionamento, criaremos o projeto *e-commerce-hibernate-search*, com as funcionalidades de um sistema tradicional de e-commerce, sendo que a busca é feita com o Hibernate Search.

É uma solução interessante para complementar a aplicação já existente. Imagine que, com Hibernate e com SQL, uma consulta pode ser ordenada pelas colunas disponíveis nas tabelas, mas isso pode não ser suficiente em todas as situações. Com o Hibernate Search podemos usar, além do SQL, as funcionalidades do Lucene:

- Ordenação de resultados por relevância, e não apenas por coluna;
- Busca por proximidade, mais eficiente que uma consulta com o **LIKE** do SQL;
- Agrupamento com *facets*;

- Geolocalização;
- Particionamento, replicação, *multi-tenancy*, computação distribuída, escalabilidade e tolerância a falhas;
- Computação em nuvem;
- Suporte a JTA e JPA.

Nesta seção, vamos ver como usar o Hibernate Search como mecanismo de busca. A aplicação de exemplo está disponível no GitHub: https://github.com/masreis/e-commerce-hibernate-search/. Este repositório é similar àquele visto no capítulo anterior, com a diferença de que agora temos o Hibernate Search como mecanismo para busca e não mais o Lucene. As classes de entidade serão as mesmas de antes: `Categoria`, `Cliente`, `ItemVenda`, `Produto` e `Venda`.

O arquivo de configuração do Maven (`pom.xml`) deve apontar para a configuração do Hibernate Search e não precisa de nenhuma dependência do Lucene.

```
<dependency>
    <groupId>org.hibernate</groupId>
    <artifactId>hibernate-search-orm</artifactId>
    <version>5.7.0.Final</version>
</dependency>
```

A versão mais recente do Hibernate Search não usa a versão mais atual do Lucene. Na versão do Hibernate Search utilizada nestes exemplos (5.7.0.Final), a versão do Lucene é a 5.5.4, bem anterior à que usamos nos outros projetos. Não chega a ser um problema, até porque o Hibernate Search não se propõe a ser um substituto para o Lucene.

O projeto do Hibernate Search oferece uma alternativa para a criação de soluções de busca escritas em Java e JPA, e como tal apresenta vantagens e desvantagens, que serão descritas ao longo do capítulo.

As vantagens estão relacionadas com a facilidade e simplicidade de criar sistemas básicos de busca, pois a base é construída em cima

do JPA, uma API amplamente conhecida e utilizada. Do outro lado, as desvantagens envolvem a dificuldade de personalização para soluções complexas.

8.1 Configuração do projeto

Para que o nosso projeto use o Hibernate Search, temos de fazer algumas alterações no arquivo `persistence.xml`, onde serão adicionadas duas propriedades. A primeira é o `indexBase`, que indica o diretório base do índice. Sob esse diretório serão criados os índices automaticamente pelo Hibernate Search. As classes de entidade indexadas terão o próprio índice gravado em um diretório com o nome da própria classe, a menos que essa configuração padrão seja alterada.

A outra propriedade é o `directory_provider`, que representa o sistema de arquivos usado no índice. No nosso exemplo, será o `FSDirectoryProvider`, que é o próprio sistema de arquivos. Também é possível usar o `RAMDirectoryProvider`, um índice em memória visto no capítulo *5. Principais classes do Lucene*.

```
<! -- Configurações do Hibernate Search -->
<property name="hibernate.search.default.indexBase"
    value="/escolha-um-diretorio/indice-ecommerce" />
<property name="hibernate.search.default.directory_provider"
    value="org.hibernate.search.store.FSDirectoryProvider" />
```

8.2 Indexando dados relacionais com Hibernate Search

Para que as classes sejam indexadas, precisamos usar a anotação `@Indexed`. No código a seguir temos a alteração para a classe

`Categoria` , que agora tem suporte ao Hibernate Search. A anotação `@Field` é aplicada a cada atributo e indica que o campo é pesquisável pelo Hibernate Search, e seus parâmetros são:

- `index = Index.YES` indica que o campo é indexado;
- `analyze = Analyze.YES` mostra que o conteúdo textual será analisado;
- `store = Store.YES` diz que o valor será gravado no diretório do índice.

Perceba que são as mesmas opções do Lucene em sua forma tradicional. Nos exemplos a seguir ocultaremos parte do código que não faz parte do Hibernate Search. O código completo do projeto pode ser visto no repositório do GitHub. Vamos para a primeira classe que será indexada, a `Categoria` . Veja:

```
@Entity
@Indexed
public class Categoria {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Field(index = Index.YES, analyze = Analyze.YES,
            store = Store.YES)
    private String nome;
    // {...}
}
```

Para indexar as classes no Hibernate Search, precisamos de um utilitário específico, que é a classe `IndexadorHS` , vista a seguir. Ela guarda similaridades com o JPA, pois utiliza o `EntityManager` para gerenciar as entidades persistentes. A diferença fica por conta do `FullTextEntityManager` que, como o nome sugere, é um gerenciador de entidade para buscas textuais. A instância de `FullTextEntityManager` é obtida a partir de um `EntityManager` e deve utilizar a sintaxe `ftem = Search.getFullTextEntityManager(em)` .

```
public class IndexadorHS implements Closeable {
    private EntityManager em;
    private FullTextEntityManager ftem;

    public IndexadorHS() {
        em = JPAUtil.getInstance().getEntityManager();
        ftem = Search.getFullTextEntityManager(em);
    }

    public void indexar(Class<?>... classes)
            throws InterruptedException {
        em.getTransaction().begin();
        ftem.createIndexer(classes).startAndWait();
        em.getTransaction().commit();
    }

    public void close() {
        em.close();
    }
}
```

A classe `Cliente` tem mais alguns campos que serão indexados, porém, não serão analisados. É o caso do e-mail, que deve ser pesquisado pelo seu valor exato. Neste caso, configure a opção `analyze = Analyze.NO`. O mesmo se aplica ao campo `cpfCnpj`, um campo que precisa ser pesquisado pelo valor exato. Já vimos mais detalhes sobre campos analisados e não analisados na seção `Field` do capítulo 5. É o correspondente no Lucene ao `StringField` e `TextField`.

O `ultimoLogin`, que guarda a data mais recente em que o usuário acessou o sistema, precisa da anotação `@DateBridge(resolution = Resolution.MINUTE, encoding = EncodingType.STRING)`, guardando a precisão até o nível de minutos e ignorando os segundos. Novamente, a data usa `analyze = Analyze.NO`, porque este campo não deve ser analisado.

Por fim, o *encoding* indica que a indexação deve guardar o valor em formato String, porque o padrão é guardar em formato numérico. O

`EncodingType.STRING` é usado para gravar o campo com o formato `yyyyMMddHHmmss`. O padrão para este parâmetro é `EncodingType.NUMERIC`, que converte as informações de data para um valor do tipo `Long`. Neste caso, a consulta deveria ser feita por meio de uma `NumericRangeQuery`, visto mais para a frente no capítulo.

Um ponto interessante é que o identificador desta entidade é o campo `Id`, então, o Hibernate Search sabe disso e usa o mesmo campo para identificar o documento indexado. Se quiser usar um campo diferente para identificar o documento, como o CPF ou CNPJ da pessoa, use a anotação `@DocumentId` sobre o atributo desejado.

O valor padrão dos parâmetros da anotação `@Field` é `index = Index.YES, analyze = Analyze.YES, store = Store.NO`, ou seja, só é necessário especificar na classe quando os valores forem diferentes destes. Na próxima classe, que é a `Cliente`, vamos escrever apenas o `store = Store.YES`, que não faz parte do padrão. Assim, a classe fica dessa forma:

```
@Entity
@Indexed
public class Cliente {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Field(analyze = Analyze.NO, store = Store.YES)
    private String email;

    @Field(store = Store.YES)
    private String nome;

    @Field(analyze = Analyze.NO, store = Store.YES)
    @DateBridge(resolution = Resolution.SECOND,
            encoding = EncodingType.STRING)
    private Date ultimoLogin;

    @Field(store = Store.YES)
```

```
    private String cpfCnpj;
    // {...}
}
```

A indexação inicial será feita com a classe `IndexadorHSTest`. Perceba que o método `indexar` admite uma lista de zero ou mais classes como parâmetro. Caso não informe nenhuma, todas as classes mapeadas pelo Hibernate Search serão indexadas. No nosso teste, estamos passando apenas a classe `Categoria` para indexação, assim, todos os registros dessa entidade serão indexados. Em tabelas muito grandes, essa operação pode demorar. Futuramente, quando o sistema estiver funcionando, o próprio Hibernate Search fará automaticamente as operações de CRUD (inclusão, consulta, atualização e exclusão) no índice.

```
public class IndexadorHSTest {
    private static IndexadorHS indexador;

    @BeforeClass
    public static void inicializar() {
        indexador = new IndexadorHS();
    }

    @AfterClass
    public static void finalizar() {
        indexador.close();
    }

    @Test
    public void testIndexarCategoria()
            throws InterruptedException {
        indexador.indexar(Categoria.class);
    }
    // {...}
}
```

Para continuar a indexação do resto do sistema, vamos indexar as classes `Cliente`, `Produto` e `Venda`, usando novamente o

`IndexadorHSTest` . Adicione esses novos métodos na classe `IndexadorHSTest` :

```
@Test
public void testIndexarCliente()
        throws InterruptedException {
    indexador.indexar(Cliente.class);
}

@Test
public void testIndexarProduto()
        throws InterruptedException {
    indexador.indexar(Produto.class);
}

@Test
public void testIndexarVenda() throws InterruptedException {
    indexador.indexar(Venda.class);
}
```

Neste ponto temos a estrutura básica da nossa aplicação com os dados carregados. A seguir, faremos as consultas com o Hibernate Search.

8.3 Consultando com Hibernate Search

Depois de indexar os registros, podemos executar as consultas. Para facilitar a leitura, vamos começar com uma pequena classe de teste, a `CategoriaTest` , que será dividida em duas partes para separar as funcionalidades e facilitar o entendimento.

Na primeira parte vamos ver a inicialização das variáveis. Para usar o Hibernate Search precisamos de um `EntityManager` e de um `FullTextEntityManager` , que será responsável por realizar as buscas no índice. Veja como criar essas variáveis:

```
public class CategoriaTest {
    private static EntityManager em;
    private static FullTextEntityManager ftem;

    @BeforeClass
    public static void inicializar() {
        em = JPAUtil.getInstance().getEntityManager();
        ftem = Search.getFullTextEntityManager(em);
    }

    @AfterClass
    public static void finalizar() {
        em.close();
    }
    // {...}
}
```

O segundo passo é executar a busca no índice. Começaremos por criar um `QueryBuilder` para a entidade `Categoria` com o comando `QueryBuilder qb = ftem.getSearchFactory().buildQueryBuilder().forEntity(Categoria.class).get()`. A `Query` retornará os itens que contêm a palavra `jogos` no campo `nome`.

Com o método `ftem.createFullTextQuery(query, Categoria.class)` o Hibernate Search vai criar um objeto `FullTextQuery`, onde cada um dos itens encontrados pela busca será convertido em uma `Categoria`. A partir deste ponto, o resultado da consulta apresenta o mesmo comportamento de uma aplicação JPA tradicional, ou seja, depois da consulta o JPA carrega o objeto `Categoria` com os dados recuperados.

> As buscas no Hibernate Search usam o conceito de API fluente, ou seja, a API expõe as opções de mapeamos de forma intuitiva. Dessa forma, o recurso de autocompletar da sua IDE vai mostrando as opções disponíveis à medida que você digita os comandos. Neste link você pode conferir mais sobre o assunto: https://dzone.com/articles/java-fluent-api-design/.

A seguir, veremos uma sequência de exemplos com consultas variadas usando Hibernate Search. O primeiro exemplo mostra a busca clássica por palavra-chave que encontra as categorias que têm *xbox* no nome:

```
@Test
public void testBuscaPeloNome() {
    // QueryBuilder
    QueryBuilder qb =
            ftem.getSearchFactory().buildQueryBuilder()
                    .forEntity(Categoria.class).get();
    // Query
    Query query = qb.keyword().onField("nome")
            .matching("xbox").createQuery();
    FullTextQuery ftQuery =
            ftem.createFullTextQuery(query, Categoria.class);
    // Resultado da consulta
    List<Categoria> lista = ftQuery.getResultList();
    Assert.assertTrue(lista.size() > 0);
    for (Categoria c : lista) {
        System.out.println(c.getNome());
    }
}
```

O método `testBuscaPorFrase`, mostrado adiante, encontra os produtos que têm a frase *xbox one* no nome, desta forma: `Query query = qb.phrase().onField("nome").sentence("xbox one").createQuery()`. Vamos ocultar as linhas repetidas e mostrar apenas o código diferente.

```
Query query = qb.phrase().onField("nome")
        .sentence("xbox one").createQuery();
```

A busca por intervalo de data usa o campo `ultimoLogin` na classe `Cliente`, que foi indexada com a precisão `Resolution.SECOND`. Neste caso, os dados usam o formato `yyyyMMddHHmmss`. No bloco de código a seguir, vemos um exemplo que verifica os clientes que acessaram o sistema entre as datas de início (`2017-09-10`) e fim (`2017-09-11`). Para consultar campos do tipo data que usam `@DateBridge`, como o

`ultimoLogin`, use o método `ignoreFieldBridge`. O código da busca está na classe `ClienteTest`.

Observação: a data segue padrões globais chamados de *time zones* e que as configurações do Hibernate, do MySQL e da sua aplicação podem ter zonas diferentes. Se as consultas por intervalo não estiverem funcionando corretamente, verifique o *time zone* no servidor e no banco de dados.

```
String dataInicio = "20170910000000";
String dataFim = "20170911000000";
Query query = qb.range().onField("ultimoLogin")
        .ignoreFieldBridge().from(dataInicio).to(dataFim)
        .createQuery();
```

Para consultar vários campos, usamos o método de combinação `bool()`, como visto no próximo bloco, onde as consultas `queryEmail` e `queryCpf` são agrupadas e formam uma única consulta com o objetivo de encontrar os clientes com o e-mail e o CPF indicados. Separamos em três variáveis apenas para facilitar a visualização do código, mas, geralmente, a instrução é escrita em uma única linha, com o uso da API chamada de fluente.

```
Query queryEmail = qb.keyword().onField("email")
    .matching("fulano@abc.net").createQuery();
Query queryCpf = qb.keyword().onField("cpfCnpj")
    .matching("1807802635").createQuery();
Query query = qb.bool().must(queryEmail).must(queryCpf)
    .createQuery();
```

A busca por disjunção é o inverso da busca por combinação. Aqui, um dos campos deve obrigatoriamente aparecer no resultado, enquanto o outro não deve aparecer quando estiver marcado com o método `not()`. No código a seguir, o resultado deve conter o e-mail indicado, entretanto, não deve incluir o CPF. Esta é a sintaxe:

```
Query query = qb.bool().must(queryEmail).must(queryCpf)
    .not().createQuery();
```

A busca com o método `should` usa a junção entre as duas consultas com o operador lógico OU, assim, o resultado mostra documentos que atendem a qualquer um dos critérios.

```
Query queryEmail = qb.keyword().onField("email")
        .matching("fulano@abc.net")
        .createQuery();
Query queryCpf = qb.keyword().onField("cpfCnpj")
        .matching("1962149214").createQuery();
Query query = qb.bool().should(queryEmail)
        .should(queryCpf).createQuery();
```

Para usar os *wild cards*, ou curingas (`?` e `*`), adicione o método `wildcard()` na sua consulta, lembrando que o `?` substitui um único caractere, enquanto o `*` substitui uma sequência deles.

```
Query query = qb.keyword().wildcard().onField("nome")
        .matching("marc*").createQuery();
// ou
Query query = qb.keyword().wildcard().onField("nome")
        .matching("marc?").createQuery();
```

O Hibernate Search também tem a busca `fuzzy` , ou busca por termo impreciso, da mesma forma como visto no capítulo 4. Para usá-la, temos os métodos `fuzzy` , `withEditDistanceUpTo` e `withPrefixLength` . A distância pode assumir os valores 1 e 2, para indicar maior ou menor precisão, e o prefixo pode assumir valores iguais ou maiores que zero. Para maiores detalhes, volte ao capítulo *4. Tipos de busca*.

```
Query query = qb.keyword().fuzzy()
        .withEditDistanceUpTo(1).onField("nome")
        .matching("xbox").createQuery();
```

Outra busca interessante é por proximidade de palavras. No exemplo a seguir, são retornados os itens que contêm os termos *xbox* e *war* com até 3 palavras de distância.

```
Query query = qb.phrase().withSlop(3). onField("nome")
        .sentence("xbox war").createQuery();
```

Para finalizar, o analisador padrão pode ser modificado com a anotação `@Analyzer`. O analisador padrão foi construído para o idioma inglês, mas há um para o português do Brasil, como vimos no capítulo 5. O código adiante mostra como usá-lo. Por ser uma mudança na estrutura da classe, o índice deve ser reconstruído com a classe `IndexadorHSTest`.

```
@Analyzer(impl = BrazilianAnalyzer.class)
public class Produto {
    // {...}
}
```

Vimos um modelo de como indexar e consultar usando o Hibernate Search e a sua API fluente. Nas próximas seções serão apresentados outros exemplos usando situações mais específicas, como a indexação de números e de campos binários como PDFs e DOCs.

8.4 Indexando campos numéricos

O Hibernate Search identifica o tipo de dado de cada campo, e tenta fazer o melhor possível nesta operação. No caso dos campos String é usado o tradicional índice invertido. Para campos numéricos (`Byte`, `Short`, `Integer`, `Long`, `Double` e `Float`), o Hibernate Search usa outra estrutura de dados, chamada de **Trie** (https://www.toptal.com/java/the-trie-a-neglected-data-structure/), que é um tipo de árvore de busca otimizado para busca por prefixos.

Entretanto, para casos específicos, precisamos usar algumas outras anotações. O atributo `preco`, por exemplo, é do tipo `BigDecimal` para poder armazenar valores monetários com precisão de duas casas decimais, ou seja, os valores são gravados no formato `1.23`. Vamos usar as anotações `@NumericField` para identificar que é um campo numérico e `@FieldBridge` para informar a classe responsável pela

indexação deste campo. Esse bloco está na classe `Produto`, vista a seguir:

```
public class Produto {
    // {...}
    @Field(store = Store.YES)
    @NumericField
    @FieldBridge(impl = BigDecimalNumericFieldBridge.class)
    private BigDecimal preco;
    // {...}
}
```

Em `BigDecimalNumericFieldBridge` fazemos a conversão de `BigDecimal` para `Double`. Essa conversão é necessária porque podemos indexar campos do tipo `Double`, mas não `BigDecimal`. Como vimos, o Hibernate Search permite a indexação de `Byte`, `Short`, `Integer`, `Long`, `Double` e `Float`, sendo que `Byte` e `Short` são indexados como `Integer`. A conversão é feita no método `set`, onde o valor do `BigDecimal` é convertido em `Double`. Os demais métodos, `get`, `objectToString` e `configureFieldMetadata`, informam à biblioteca o tipo de dado deste campo. Veja:

```
public class BigDecimalNumericFieldBridge implements
        MetadataProvidingFieldBridge, TwoWayFieldBridge {

    @Override
    public void set(String name, Object value, Document document,
            LuceneOptions luceneOptions) {
        if (value != null) {
            BigDecimal decimalValue = (BigDecimal) value;
            Double indexedValue = decimalValue.doubleValue();
            luceneOptions.addNumericFieldToDocument(name,
                    indexedValue, document);
        }
    }

    @Override
    public Object get(String name, Document document) {
        String fromLucene = document.get(name);
        BigDecimal storedBigDecimal = new BigDecimal(fromLucene);
```

```
        return storedBigDecimal;
    }

    @Override
    public String objectToString(Object object) {
        return object.toString();
    }

    @Override
    public void configureFieldMetadata(String name,
            FieldMetadataBuilder builder) {
        builder.field(name, FieldType.DOUBLE);
    }
}
```

Para consultar por este campo, precisamos usar o método `NumericRangeQuery.newDoubleRange`, alterando para tipo numérico adequado ao campo consultado. Além do `Double`, há opção para `Float`, `Int` e `Long`. Os testes de Produto estão na classe `ProdutoTest`. Vejamos o primeiro:

```
    @Test
    public void testBuscaPorIntervaloNRQ() {
        NumericRangeQuery<Double> query =
                NumericRangeQuery.newDoubleRange("preco",
                        precoMinimo, precoMaximo, true, true);
        FullTextQuery ftQuery =
                ftem.createFullTextQuery(query, Produto.class);
        List<Produto> lista = ftQuery.getResultList();
        Assert.assertTrue(lista.size() > 0);
        for (Produto p : lista) {
            System.out
                    .println(p.getNome() + " - " + p.getPreco());
        }
    }
```

Há mais duas formas de fazer a mesma consulta. A primeira é com uma consulta booleana combinando duas outras consultas por `range`. Dessa forma:

```
QueryBuilder qb =
        ftem.getSearchFactory().buildQueryBuilder()
                .forEntity(Produto.class).get();
Query queryAbove = qb.range().onField("preco")
        .above(precoMinimo).createQuery();
Query queryBelow = qb.range().onField("preco")
        .below(precoMaximo).createQuery();
Query query = qb.bool().must(queryAbove).must(queryBelow)
        .createQuery();
```

A outra forma com uma consulta por intervalo com `from` e `to`. Assim:

```
QueryBuilder qb =
        ftem.getSearchFactory().buildQueryBuilder()
        .forEntity(Produto.class).get();
Query query = qb.range().onField("preco")
        .from(precoMinimo).to(precoMaximo).createQuery();
```

Esta seção mostrou características um pouco mais avançadas do Hibernate Search, como a `Bridge` e as consultas combinadas. Perceba que estamos revendo as funcionalidades do Lucene, desta vez usando a API do Hibernate Search.

8.5 Indexando associações entre classes

A associação entre entidades, representada pelas anotações do JPA `@ManyToMany`, `@OneToMany` e `@ManyToOne`, é indexada por meio da anotação `@IndexedEmbedded` do Hibernate Search. A classe `Produto` tem um modelo desse tipo de solução. No caso de produto, cada item pode ter uma ou várias categorias. Com essa funcionalidade, é possível pesquisar pela entidade principal e todas as entidades que estão associadas. Veja na classe `Produto` como atualizar o atributo `categorias`:

```
public class Produto {
    // {...}
```

```
    @IndexedEmbedded
    private Set<Categoria> categorias =
        new HashSet<Categoria>(0);
    // {...}
}
```

A consulta é feita com o campo `categorias.nome`, onde `categorias` é o atributo da classe `Produto` e `nome` é o atributo associado que está na classe `Categoria`. Para exemplificar, no código a seguir são recuperados os produtos que estão na categoria *periféricos*. Uma observação é que devemos usar o `ignoreFieldBridge()` quando construímos a consulta com associação:

```
Query query = qb.keyword().onField("categorias.nome")
        .ignoreFieldBridge().matching("periféricos")
        .createQuery();
```

A indexação das associações é um recurso que facilita o desenvolvimento do sistema de busca, é uma daquelas vantagens que citamos no começo do capítulo. Este recurso pode ser usado na classe `Venda`, que tem uma associação com `Clilente`, ou seja, é possível pesquisar as vendas de um cliente usando o `@IndexedEmbedded`.

8.6 Indexando campos binários

Campos binários são aqueles que contêm arquivos PDF, DOC, XLS e afins. A mesma classe `Produto` mostra como indexar um campo deste tipo, que é armazenado como um *array* de *bytes* (`byte[]`). Além de `byte[]`, é possível processar campos do tipo String, `URI` ou `java.sql.Blob` com a anotação `@TikaBridge`, que faz parte da biblioteca Tika. Para usá-la, adicione a anotação no campo correspondente:

```
public class Produto {
    // {...}
    @TikaBridge
    private byte[] especificacaoFabricante;
    // {...}
}
```

A consulta deste tipo de campo é igual às consultas anteriores e você pode usar os mesmos recursos vistos no capítulo. Este exemplo busca por produtos que têm a palavra *xbox* em sua especificação.

```
    Query query = qb.keyword()
            .onField("especificacaoFabricante")
            .ignoreFieldBridge().matching("xbox")
            .createQuery();
```

O `TikaBridge` é um utilitário para a extração do conteúdo textual de arquivos baseado no Apache Tika, que foi visto no capítulo 3. Com ele, fechamos o capítulo sobre o Hibernate Search.

Resumo

Este capítulo mostrou uma estratégia para adotar o Hibernate Search ORM em projetos escritos em Java que utilizam JPA e Hibernate. Com o Hibernate Search é possível reutilizar as classes de entidade da aplicação JPA e apenas adicionar a anotação `@Indexed`. Com isso, os registros serão indexados com o Lucene.

A partir daí, a classe será persistida no banco de dados ao mesmo tempo em que é gravada no índice do Lucene. Durante a consulta, você pode escolher se quer consultar no banco de dados relacional (com JPA) ou no índice (com o Hibernate Search). Como visto nos testes, a busca pelo índice costuma ser mais eficiente e flexível, economizando recursos do servidor de banco de dados. No entanto, o Hibernate Search é limitado em termos de funcionalidade, além de que a sua personalização pode ser mais complexa. Isso sem contar que só pode ser usado em projetos com Hibernate e JPA.

No próximo capítulo veremos funcionalidades e técnicas avançadas para indexação e busca, que permitem executar análise de texto, buscas facetadas, sugestão de resultados (aquela funcionalidade do "você quis dizer"), *highlight* de texto, união de índices, busca geoespacial e um novo tipo de busca por similaridade. Também serão mostrados utilitários que não fazem parte do projeto Lucene, mas que ajudam na administração e manutenção do índice.

Capítulo 9
Recursos avançados

O Lucene nasceu com a proposta de ser uma biblioteca para busca textual, mas a necessidade dos usuários está além das funcionalidades básicas há tempos. Para acompanhar essa evolução, o Lucene oferece pacotes complementares com recursos avançados.

Essas extensões incluem recursos como a busca com sinônimos, cálculo da frequência dos termos (para análise de texto), indexação de vetores, corretor ortográfico, sugestão de resultados, consulta *More Like This*, marcador de texto *highlighter* e o uso de *facets*. Veremos a seguir como implementar cada caso. O primeiro item a ser detalhado é o uso dos sinônimos.

9.1 Sinônimos

Sinônimos são expressões com significado igual ou aproximado. É o caso de carro, automóvel e veículo. Não são exatamente a mesma coisa, mas o sentido é próximo e, geralmente, podem ser usados como sinônimos. O sistema de busca deve ser flexível a ponto de entender o que o usuário está procurando, e os sinônimos representam uma boa opção para isso. Dessa forma, um bom dicionário de sinônimos pode melhorar a qualidade da busca. Até o nosso Java tem um caso confuso. O Java EE já foi chamado de J2EE, JavaEE e até mesmo de JEE, referindo-se ao mesmo recurso.

Há duas soluções para o problema dos sinônimos: indexar os documentos com os sinônimos, no *index time*; e buscar no índice através de sinônimos, no *query time*. A primeira opção é ruim

porque limita as opções de busca, principalmente a questão de busca por proximidade; e também porque novos sinônimos exigem a reindexação completa. A melhor solução é usar os sinônimos durante a busca, no *query time*. É o que faz a classe `AnalisadorSinonimos`.

O método `createComponents(String fieldName)` define a política para extração das palavras de um texto. O nosso `AnalisadorSinonimos` define que, durante a busca, as palavras serão substituídas pelos seus sinônimos. No caso de uma busca pelo termo `java ee`, o analisador vai considerar os termos `java ee`, `javaee`, `j2ee` e `jee`. Se buscar por `pen drive`, o buscador vai considerar `pen drive` e `pendrive`.

Note que são usados 3 filtros: o `LowerCaseFilter` transforma todas as letras em minúsculas, o `StandardFilter` separa as palavras do texto e o último, o `SynonymGraphFilter`, faz a substituição dos sinônimos.

```
public class AnalisadorSinonimos extends StopwordAnalyzerBase {
    private static final Logger logger =
            Logger.getLogger(AnalisadorSinonimos.class);
```

```
public class AnalisadorSinonimos extends StopwordAnalyzerBase {
    private static final Logger logger =
            Logger.getLogger(AnalisadorSinonimos.class);

    @Override
    protected TokenStreamComponents createComponents(
            String fieldName) {
        // Cria o analisador com o filtro de sinônimos
        boolean ignoraMaiusculas = true;
        Tokenizer tokenizer = new StandardTokenizer();
        TokenStream filtro = new LowerCaseFilter(tokenizer);
        filtro = new StandardFilter(filtro);
        filtro = new SynonymGraphFilter(filtro,
                criaMapaSinonimos(), ignoraMaiusculas);
        return new TokenStreamComponents(tokenizer,
                criaTokenStream(filtro));
```

```
    }
}
```

O próximo método é `criaMapaSinonimos`, onde é carregado o arquivo com a lista de sinônimos. As opções são deduplicação e expansão dos sinônimos. A deduplicação remove os sinônimos repetidos e a expansão significa que o analisador deve considerar todos os sinônimos como sendo termos semelhantes. Por exemplo: `pendrive` e `pen drive` serão tratados como termos semelhantes. Se não for usada a expansão, toda ocorrência de `pendrive` será substituída por `pen drive`, de forma que a palavra `pendrive` será desconsiderada. O arquivo de sinônimos é carregado pelo `SolrSynonymParser` e no final é retornado um mapa com os sinônimos através do `SynonymMap`.

```
    private SynonymMap criaMapaSinonimos() {
        boolean fazDeduplicacao = true;
        boolean expandeListaSinonimos = true;
        SolrSynonymParser parser = new SolrSynonymParser(
                fazDeduplicacao, expandeListaSinonimos,
                criaAnalizadorSinonimos());
        try {
            parser.parse(new FileReader(
                    "src/test/resources/sinonimos.txt"));
            return parser.build();
        } catch (IOException | ParseException e) {
            logger.error(e);
            throw new RuntimeException(e);
        }
    }
```

Veja um modelo do arquivo `sinonimos.txt`:

```
tb,tib,terabyte,terabytes
gb,gib,gigabytes
mb,mib,megabytes
Televisao,Televisor,TV,TVs
pendrive,pen drive
veículo,carro,moto,bicicleta
xone,xbox one
hd,disco rígido
```

```
jee,java ee,javaee,j2ee
jogo,game
wii,nintendo
```

O arquivo de sinônimos é bem simples. Cada linha contém uma lista de sinônimos que devem estar separados por vírgulas. Para forçar a substituição de uma palavra, use a sintaxe `pendrive=>pen drive`. Toda ocorrência de `pendrive` será apenas substituída por `pen drive`.

Os últimos métodos são o `criaTokenStream` e o `criaAnalisadorSinonimos`. O `criaTokenStream` é usado para retornar um `TokenStream`, que é a classe que implementa as regras de separação das palavras. Para índices criados com o `BrazilianAnalyzer`, usamos o `BrazilianStemFilter`:

```
    private TokenStream criaTokenStream(TokenStream filtro) {
        return new BrazilianStemFilter(filtro);
    }

    private Analyzer criaAnalizadorSinonimos() {
        return new BrazilianAnalyzer();

    }
```

Diferente do exemplo anterior, para índices criados com o `StandardAnalyzer`, apenas retorne o próprio filtro que já é do tipo padrão. O `criaAnalisadorSinonimos` também deve acompanhar esta regra e retorna um `StandardAnalyzer`:

```
    private TokenStream criaTokenStream(TokenStream filtro) {
        return filtro;
    }

    private Analyzer criaAnalizadorSinonimos() {
        return new StandardAnalyzer();

    }
```

Em `AnalisadorSinonimosTest` temos vários testes para verificar o comportamento do analisador de sinônimos. A busca por `8`

`gigabytes` encontra um sinônimo e faz as substituições indicadas no arquivo `sinonimos.txt`.

```
Analyzer analyzer = criaAnalisador();
QueryParser parser = new QueryParser("", analyzer);
String consulta = "produtoNome:(\"8 gigabytes\")";
Query query = parser.parse(consulta);
logger.info("Consulta analisada-> " + query);
```

O método `criaAnalisador` usado para criar o *parser* deve retornar um `AnalisadorSinonimos`:

```
private Analyzer criaAnalisador() {
    return new AnalisadorSinonimos();
}
```

A consulta executada pelo *parser* é `produtoNome:8 Synonym(produtoNome:gb produtoNome:gib produtoNome:gigabyt)`. Perceba que depois do número 8 aparece a palavra *Synonym*, seguida da lista de sinônimos para *gigabyte*. Veja agora o caso da consulta por `pendrive`: `((+produtoNome:pen +produtoNome:driv) produtoNome:pendriv)`. O *parser* analisa de uma forma diferente. Por fim, a busca por `\"disco rígido\"`, ou seja, um termo com mais de uma palavra, resulta em `spanOr([produtoNome:hd, spanNear([produtoNome:disc, produtoNome:rig], 0, true)])`.

Para concluir, vou deixar uma questão para você pensar. Em sistemas especialistas, os sinônimos podem variar. Sistemas médicos, jurídicos, de transporte e de engenharia têm dicionários próprios. Esta é uma área de pesquisa complexa e não existe uma solução perfeita, mas, sim, soluções com uma boa precisão. Ainda pensando no exemplo do carro, automóvel e veículo, já falamos que são sinônimos, entretanto, em alguns contextos isso não é verdade. Veja: veículo é qualquer meio de transporte, mas não necessariamente é um automóvel, pois este precisa de autopropulsão (motor). Assim, não é totalmente verdade que todo veículo é um automóvel. Um vagão de trem e uma bicicleta (geralmente) não têm motor.

Fica claro que a própria aquisição do dicionário de sinônimos é um desafio, porque é necessário que um especialista na área liste estes sinônimos. Por exemplo, um médico, advogado ou engenheiro precisaria listar os sinônimos e cadastrar no sistema de busca. Por uma questão didática, nosso exemplo conta com uma lista limitada de sinônimos.

9.2 Frequência dos termos

O índice do Lucene é composto por documentos, que por sua vez são compostos por palavras (ou termos). Neste cenário, alguns termos são mais relevantes que outros, como foi explicado na seção sobre TF/IDF. Para explorar o índice, uma opção é estudar a frequência dos termos, isto é, verificar quantas vezes uma palavra aparece em cada documento e o seu total de ocorrências em todo o índice.

Mesmo em grandes sistemas de inteligência artificial é necessário conhecer a informação que será usada, o que é feito através de uma análise exploratória, sem grandes pretensões. O objetivo é apenas ver como os dados estão dispostos na base. E o Lucene oferece recursos interessantes neste sentido. Em um sistema de busca, esse estudo é importante para melhorar a qualidade dos resultados, removendo dados inúteis ou errados da base. Com essa abordagem, podemos descobrir termos sem relevância que estão diminuindo a precisão das buscas. É o caso das *stop words*, que foram vistas no capítulo *2. Conceitos de recuperação da informação*, mas não se resume a apenas isso, como veremos ao longo da seção em métodos de teste que estão no projeto `e-commerce`, na classe `AnalisadorTermosIndiceTest`.

O Lucene dispõe de um utilitário para listar os termos do índice, que será o ponto de partida para a análise do índice. Este utilitário, o `MultiFields`, pode recuperar as palavras que fazem parte de um

índice e funciona como um dicionário, ou seja, ele tem a lista de todas as palavras conhecidas naquele índice. Para criar esse dicionário, apenas passe o `IndexReader` e o nome do campo que quer analisar. Ele retorna um `Iterator` que pode ser impresso no console, como neste bloco:

```
public void testMostraTermosIndice() throws IOException {
    String campo = "produtoNome";
    Terms termos = MultiFields.getTerms(reader, campo);
    TermsEnum iteTermos = termos.iterator();
    BytesRef next;
    while ((next = iteTermos.next()) != null) {
        System.out.println(next.utf8ToString());
    }
}
```

Para analisar um termo específico, por exemplo, a consulta `produtoDescricao:xbox` , o próprio `IndexReader` oferece os métodos `totalTermFreq(Term)` e `docFreq(Term)` . Eles mostram o `Term Frequency (TF)` e o `Document Frequency (DF)` , usados no cálculo do TF/IDF. O TF é a quantidade de ocorrências do termo no índice e o DF é a quantidade de documentos que contêm o termo. Veja no código:

```
public void testDadosTermo() throws IOException {
    Term termo = new Term("produtoDescricao", "xbox");
    String message = String.format(
            "%s:%s \nTotal de Ocorrências (TF): %3$d\n"
                    + "Total de Documentos (DF):%4$d",
            termo.field(), termo.text(),
            reader.totalTermFreq(termo),
            reader.docFreq(termo));
    System.out.println(message);
}
```

O resultado desse teste, no meu índice local, mostra que a palavra `xbox` aparece 669 vezes no campo `produtoDescricao` , como visto na listagem a seguir. Além disso, a palavra `xbox` aparece em 481 produtos diferentes.

```
produtoDescricao:xbox
Total de Ocorrências (TF): 669
Total de Documentos (DF):481
```

Combinando esses dois testes, podemos mostrar os totais para todos os termos do campo `produtoNome` no índice. Dessa forma:

```
    public void testMostraTermosETotais() throws IOException {
        String campo = "produtoNome";
        Terms termos = MultiFields.getTerms(reader, campo);
        TermsEnum iteTermos = termos.iterator();
        BytesRef next;
        while ((next = iteTermos.next()) != null) {
            String palavra = next.utf8ToString();
            Term termo = new Term(campo, palavra);
            String message = String.format(
                    "%s:%s - TF[%3$d] DF[%4$d]", termo.field(),
                    termo.text(), reader.totalTermFreq(termo),
                    reader.docFreq(termo));
            System.out.println(message);
        }
    }
```

Uma parte da listagem resultante está logo a seguir, na qual podemos conferir que há termos com muitas ocorrências, como em `universal` que tem 90 ocorrências, e `university` que tem apenas 1 ocorrência. Veja que em `universitari` temos uma palavra que foi analisada pelo `BrazilianAnalyzer`, um analisador estudado no capítulo *5. Principais classes do Lucene*. O último termo é `xbox360`, e isso significa que o nome do produto foi cadastrado errado. O correto seria `xbox 360`. Temos técnicas de processamento de texto para ajudar a resolver esse problema.

```
produtoNome:universal - TF[90] DF[90]
produtoNome:universitari - TF[1] DF[1]
produtoNome:xbox360 - TF[2] DF[2]
```

9.3 Indexando campos com vetores

O Lucene permite o armazenamento de informações adicionais sobre cada palavra indexada, na forma de um vetor de termos (*term vector*). Além do vetor, existe a opção de armazenar outros dados sobre os termos, que serão usados mais à frente em funcionalidades como o *highlighter*. São estas as opções:

- `setStoreTermVectors` : armazena as palavras do documento em um vetor;
- `setStoreTermVectorOffsets` : guarda a posição de início e fim de cada palavra no documento;
- `setStoreTermVectorPayloads` : o *payload* é uma informação adicional sobre o termo, indicando que ele tem um peso diferente dos demais;
- `setStoreTermVectorPositions` : guarda a ordem em que o termo aparece no vetor de termos do documento.

Para ilustrar como usar o vetor de termos, vamos criar um campo no índice que chamaremos de `produtoDescricaoVetor` . Ele funciona como um campo indexado tradicional, assim, em um sistema real, você pode usar tanto o `produtoDescricaoVetor` quanto o próprio `produtoDescricao` , não seria necessário ter os dois.

Essas configurações estão em `IndexadorProduto` , de forma que precisamos reindexar os produtos para essa mudança ter efeito. O trecho importante aqui é o `tipoComPosicoes.setStoreTermVectors(true)` , que criará um vetor com as palavras do documento. As outras opções serão exploradas no decorrer do capítulo.

Considerando o novo campo e que estamos armazenando mais informações sobre os campos, o tamanho do índice aumentará sensivelmente. Veja o código:

```
private void preencherDescricaoVetor(Produto produto,
        Document doc) {
    FieldType tipoComPosicoes = new FieldType();
```

```
        tipoComPosicoes.setIndexOptions(
                IndexOptions.
                DOCS_AND_FREQS_AND_POSITIONS_AND_OFFSETS);
        tipoComPosicoes.setStored(true);
        tipoComPosicoes.setStoreTermVectorOffsets(true);
        tipoComPosicoes.setStoreTermVectorPayloads(true);
        tipoComPosicoes.setStoreTermVectorPositions(true);
        tipoComPosicoes.setStoreTermVectors(true);
        String descricao = produto.getDescricao();
        if (descricao == null) {
            descricao = produto.getNome();
        }
        doc.add(new Field("produtoDescricaoVetor", descricao,
                tipoComPosicoes));
    }
```

A análise dos termos fica no método `testAnalisaTermosDocumento` da classe `AnalisadorTermosIndiceTest`. O ponto central é o método `IndexReader.getTermVector(docId, campo)` que retorna o vetor de termos para um campo em um documento (`docId`) específico. Vale lembrar que o campo analisado deve ser indexado com a opção `setStoreTermVectors(true)`.

```
    public void testAnalisaTermosDocumento()
            throws IOException {
        for (int docId = 0; docId < reader.maxDoc(); docId++) {
            // Imprime nome do arquivo
            Document documento = reader.document(docId);
            System.out.println(
                    String.format("Analisando produto [%s]",
                            documento.get("produtoNome")));
            // Recupera termos do documento
            String campo = "produtoDescricaoVetor";
            Terms vetorTermos =
                    reader.getTermVector(docId, campo);
            if (vetorTermos == null) {
                System.err
                        .println("Não há termos no documento.");
                continue;
            }
```

```
            TermsEnum termos = vetorTermos.iterator();
            // Monta um mapa com a quantidade de
            // ocorrências de cada termo dentro do documento
            Map<String, Integer> frequencias =
                    criaMapaDeTermosEFrequencias(termos);
            // Imprime a frequência com que
            // cada termo aparece no documento
            String conteudo = frequencias.entrySet().stream()
                    .map(e -> e.getKey() + "[" + e.getValue()
                            + "]\n")
                    .collect(Collectors.joining());
            System.out.println(conteudo);
        }
    }
```

A partir do vetor de termos, criamos um mapa que contém a quantidade de ocorrências de cada palavra no documento. Ao analisar o código, vai perceber que os dados são armazenados como `bytes` no índice, através da classe `BytesRef`. Como visto antes neste capítulo, o uso de `bytes` melhora a performance das operações.

```
    private Map<String, Integer> criaMapaDeTermosEFrequencias(
            TermsEnum termos) throws IOException {
        BytesRef bytesRef = null;
        Map<String, Integer> frequencias = new TreeMap<>();
        while ((bytesRef = termos.next()) != null) {
            String termo = bytesRef.utf8ToString();
            PostingsEnum postingEnum =
                    termos.postings(null, PostingsEnum.FREQS);
            int noMoreDocs = DocIdSetIterator.NO_MORE_DOCS;
            while (postingEnum.nextDoc() != noMoreDocs) {
                int freqDocumento = postingEnum.freq();
                Integer freqMapa = frequencias.get(termo);
                if (freqMapa == null) {
                    freqMapa = 0;
                }
                freqMapa += freqDocumento;
                frequencias.put(termo, freqMapa);
            }
```

```
        }
        return frequencias;
    }
```

Como resultado, obtemos uma lista de palavras e a quantidade de vezes em que ela aparece no arquivo. A listagem a seguir mostra as palavras extraídas do jogo New Super Mario Bros U do Nintendo Wii U. Algumas palavras são relevantes, enquanto outras são irrelevantes para este contexto. Este é o primeiro passo no processo de mineração de texto (*text mining*). As duas tabelas seguintes se referem à mesma descrição do produto, entretanto, na primeira a análise é feita com o `BrazilianAnalyzer` e na outra, com o `StandardAnalyzer`. A descrição original do jogo é:

> New Super Mario Bros U é uma nova aventura side-scrolling com Mario, Luigi, Toad e até mesmo seu personagem Mii! Agora é sua chance para entrar no Reino do Cogumelo e explorar novos mundos, novos poderes e novas maneiras de jogar.

A tabela a seguir mostra os termos gerados pelo `BrazilianAnalyzer`, que usa a gramática em português do Brasil. Essa análise reduz as palavras ao seu radical e remove *stop words*. Radical é o elemento gramatical que contém o significado da palavra. Exemplo: o radical de `agora` é `agor`, o radical de `aventura` é `aventur`. As palavras `jogo` e `jogar` são reduzidas a `jog`, sendo duas ocorrências para o radical `jog`. As palavras `nova`, `novas` e `novos` são reduzidos para o radical `nov`, que tem 4 ocorrências. O caso de `wii` não é analisado porque esta palavra está em inglês.

```
<table>
    <tr>
        <th colspan="4">Termos com BrazilianAnalyzer</th>
    </tr>
    <tr>
        <td>agor[1]</td>
        <td>ate[1]</td>
        <td>aventur[1]</td>
```

```
    <td>bros[1]</td>
</tr>
<tr>
    <td>chanc[1]</td>
    <td>cogumel[1]</td>
    <td>entrar[1]</td>
    <td>explor[1]</td>
</tr>
<tr>
    <td>jog[2]</td>
    <td>luig[1]</td>
    <td>maneir[1]</td>
    <td>mari[2]</td>
</tr>
<tr>
    <td>mii[1]</td>
    <td>mund[1]</td>
    <td>new[1]</td>
    <td>nint[1]</td>
</tr>
<tr>
    <td>no[1]</td>
    <td>nov[4]</td>
    <td>par[2]</td>
    <td>personag[1]</td>
</tr>
<tr>
    <td>pod[1]</td>
    <td>rein[1]</td>
    <td>scrolling[1]</td>
    <td>sid[1]</td>
</tr>
<tr>
    <td>sup[1]</td>
    <td>toad[1]</td>
    <td>u[2]</td>
    <td>wii[1]</td>
</tr>
<tr>
    <td colspan="4">é[2]</td>
```

```
    </tr>
</table>
```

O analisador padrão do Lucene (`StandardAnalyzer`) não reduz as palavras ao seu radical, nem retira as *stop words*, que neste caso são `de` , `do` , `e` , `para` e `uma` . O vetor de termos gerado por este analisador está na tabela a seguir. Este analisador tem uma saída bem diferente, por exemplo, as palavras `jogar` e `jogo` são duas ocorrências diferentes. O caso da palavra `maneiras` é interessante. Veja que a palavra está no plural e que uma busca por `maneira` não retorna esse registro. Neste caso, o `BrazilianAnalyzer` é mais interessante, porque o radical seria `maneir` , que equivale a `maneira` , `maneiro` , `maneiras` etc.

```
<table>
    <tr>
        <th colspan="4">Termos com StandardAnalyzer</th>
    </tr>
    <tr>
        <td>agora[1]</td>
        <td>até[1]</td>
        <td>aventura[1]</td>
        <td>bros[1]</td>
    </tr>
    <tr>
        <td>chance[1]</td>
        <td>cogumelo[1]</td>
        <td>com[1]</td>
        <td>de[1]</td>
    </tr>
    <tr>
        <td>do[1]</td>
        <td>e[3]</td>
        <td>entrar[1]</td>
        <td>explorar[1]</td>
    </tr>
    <tr>
        <td>jogar[1]</td>
        <td>jogo[1]</td>
        <td>luigi[1]</td>
```

```
        <td>maneiras[1]</td>
    </tr>
    <tr>
        <td>new[1]</td>
        <td>nintendo[1]</td>
        <td>nova[1]</td>
        <td>novas[1]</td>
    </tr>
    <tr>
        <td>novos[2]</td>
        <td>mario[2]</td>
        <td>mesmo[1]</td>
        <td>mii[1]</td>
    </tr>
    <tr>
        <td>mundos[1]</td>
        <td>para[2]</td>
        <td>personagem[1]</td>
        <td>poderes[1]</td>
    </tr>
    <tr>
        <td>reino[1]</td>
        <td>scrolling[1]</td>
        <td>seu[1]</td>
        <td>side[1]</td>
    </tr>
    <tr>
        <td>sua[1]</td>
        <td>super[1]</td>
        <td>toad[1]</td>
        <td>u[2]</td>
    </tr>
    <tr>
        <td>uma[1]</td>
        <td>wii[1]</td>
        <td colspan="2">é[2]</td>
    </tr>
</table>
```

Um dos termos listados é `toad`, um personagem no universo do jogo Mario. É um item importante para encontrar este jogo, ou seja, pode

ter peso maior que as outras palavras. Agora compare a palavra `toad` com a palavra `mundo`, que faz parte do mesmo vetor. Certamente a palavra `mundo` é pouco importante para encontrar o jogo do Mario, entretanto, as duas fazem parte da mesma descrição. Fica claro que os pesos das palavras variam conforme o contexto. O contexto do jogo do Mario tem palavras significativas, o contexto de outro jogo, como Halo, teria palavras significativas diferentes.

O Google, LinkedIn e Netflix, por exemplo, guardam o histórico de navegação para melhorar a precisão dos serviços de busca. Vamos imaginar um sistema de análise curricular, como o próprio LinkedIn. Se há grande procura por SQL, um profissional especializado em Oracle poderia ser avisado para atualizar seus dados, porque, possivelmente, ele sabe SQL, mas não está sendo encontrado pelos recrutadores que procuram por SQL.

Este tipo de integração entre o usuário e o buscador cria um tipo de busca orgânica, alterando a forma como os itens são indexados e ordenados. Por padrão, o Lucene não é orgânico, pelo contrário. O Lucene considera apenas os cálculos de TF/IDF para ordenar o resultado. Agora, com essa análise textual, pode-se verificar quais os termos que têm maior relevância para os usuários e reorientar o cálculo de relevância de cada termo. Dessa forma, a ordenação do resultado seria dinâmica, porque sofreria alterações em função de parâmetros externos, que não são apenas o TF/IDF.

9.4 Corretor ortográfico (spell checker)

O Lucene tem utilitários para correção de ortografia dos dados digitados pelo usuário. Assim, quando o usuário escreve uma frase, é possível verificar se aquelas palavras têm a grafia correta fazendo uma comparação com as palavras existentes no índice. Quando há erros, o corretor pode sugerir palavras similares a partir daquelas já

indexadas. O projeto usado como exemplo será novamente o `e-commerce`, e a classe é `CorretorOrtograficoTest`.

Como esta é uma das bibliotecas complementares do Lucene, ela precisa ser adicionada através da dependência Maven `<artifactId>lucene-suggest</artifactId>`. O primeiro passo é criar o dicionário de referência. Os utilitários são `SpellChecker` e `Dictionary`. O método de criação do dicionário de referência é `indexDictionary`, com 3 parâmetros: (1) dicionário, (2) configuração e (3) *full merge* dos segmentos. Não é uma operação frequente, então, podemos usar o *full merge*.

Para criar o `SpellChecker` o único parâmetro é o diretório do índice. O dicionário usa 2 parâmetros: (1) `IndexReader` apontando para o índice e (2) o nome do campo. No nosso exemplo, vamos criar um dicionário para o campo `produtoDescricao`.

O termo com erro usado nos exemplos será *progeto*. A partir dele vamos listar as possíveis sugestões. Inicialmente, seria natural você pensar que a única correção possível é *projeto*. Mas o Lucene não sabe disso e vai sempre sugerir termos similares com base em cálculos matemáticos. Entre as sugestões vai aparecer a palavra *projeto* porque é muito parecido com o termo com erro, mas vamos lembrar de que há casos mais complicados e o Lucene deve oferecer uma solução genérica que atenda a todos.

O mecanismo funciona com base em um diretório de referência criado a partir das palavras indexadas em um determinado campo, por exemplo, `produtoDescricao`. Este dicionário contém as palavras daquele campo, com seu peso e *payload*. Para usar o corretor, siga os seguintes passos: (1) crie um dicionário de referência com as palavras de um campo do índice; (2) opcionalmente, verifique se a palavra digitada pelo usuário existe no índice; (3) o corretor retorna listas com palavras similares com base no dicionário de referência, usando técnicas de linguística computacional. Na sequência vamos ver cada um dos passos.

O primeiro passo (1) para usarmos o recurso de correção é `test1IndexaDicionario`. O `SpellChecker` vai alterar o índice ao adicionar as informações sobre o dicionário. Assim:

```
public void test1IndexaDicionario() throws IOException {
    System.out.println("Criando o índice de sugestões");
    SpellChecker corretor = new SpellChecker(diretorio);
    Dictionary dicionario =
            new LuceneDictionary(reader, campo);
    IndexWriterConfig config = new IndexWriterConfig();
    boolean fullMerge = true;
    corretor.indexDictionary(dicionario, config, fullMerge);
    corretor.close();
}
```

Agora, vamos verificar se o termo consta do dicionário (2), isto é, se a palavra *progeto* faz parte do índice. Claro que não deveria fazer, já que tem erro de grafia. Este é um passo opcional, porém, é uma funcionalidade bem interessante.

```
public void testSeExiste() throws IOException {
    SpellChecker verificador = new SpellChecker(diretorio);
    System.out.println(String.format(
            "O termo '%s' está no dicionário: %s",
            termoComErro, verificador.exist(termoComErro)));
    verificador.close();
}
```

A principal utilidade do corretor é, claro, sugerir correções (3). O `SpellChecker` tem o método `suggestSimilar`, que lista uma quantidade definida de termos similares. Nosso exemplo limita a 50 as palavras parecidas com *progeto*. Considerando o índice gerado a partir do *dump* projeto de *e-commerce*, as sugestões são *projeto*, *projetor*, *projetos*, *protetor*, *produto*, *projeta*, *promete*, *protetora* etc. São dezenas de sugestões geradas a partir de *progeto*.

Vale ressaltar que essas sugestões são geradas matematicamente e não têm nenhuma relação com seu significado. O cálculo usado

leva em consideração apenas as letras das palavras e não o seu significado.

```
public void testSugereAlternativas() throws IOException {
    System.out.println("\nSugere alternativas");
    SpellChecker corretor = new SpellChecker(diretorio);
    int quantidadeSugestoes = 50;
    String[] sugestoes = corretor.suggestSimilar(
            termoComErro, quantidadeSugestoes);
    for (String sugestao : sugestoes) {
        System.out.println(sugestao);
    }
    corretor.close();
}
```

O `SpellChecker` monta a lista de sugestões usando uma métrica chamada de Distância de Levenshtein (DL), vista anteriormente no capítulo *4. Tipos de busca*. A DL é uma forma de medir a similaridade entre palavras e representa o número de operações necessárias para transformar uma palavra em outra. A DL é o método padrão para este recurso, sendo que os outros métodos serão vistos mais à frente. No Lucene, a distância entre palavras é representada como um `float` entre 0.0 e 1.0, onde 0 significa que as palavras são totalmente diferentes e 1 significa que as palavras são iguais.

Por padrão, o Lucene usa uma medida de precisão de 0.5. Essa medida é chamada internamente de acurácia. No exemplo a seguir veremos o que acontece quando usamos acurácia (precisão) de 0.7, ou seja, pede-se uma similaridade maior entre a palavra de origem e as sugestões.

```
int quantidadeSugestoes = 5;
float acuracia = 0.7f;
String[] sugestoes = corretor.suggestSimilar(
        termoComErro, quantidadeSugestoes, acuracia);
```

O resultado com acurácia de 0.7 mostra apenas 4 itens: *projeto*, *projetor*, *projetos* e *protetor*. Perceba que são apenas 4 itens, logo

houve um filtro que retirou a grande maioria das sugestões. Quanto maior a acurácia, menor a quantidade de sugestões.

Além da DL, as outras alternativas para gerar sugestões com o `SpellChecker` são implementações de `StringDistance`, como o `JaroWinklerDistance`, `NGramDistance` e `LevensteinDistance` (o padrão). Cada uma usa um algoritmo diferente e traz resultados diferentes. Para Strings curtas, o caso mais comum, o `JaroWinklerDistance` pode ser a melhor opção. Para usar essa opção, crie `SpellChecker` com a sintaxe: `SpellChecker corretor = new SpellChecker(diretorio, criaStringDistance())`. O segundo parâmetro é o `criaStringDistance`, que indica a estratégia para medir a distância entre palavras e pode ser visto aqui:

```
private StringDistance criaStringDistance() {
    return new JaroWinklerDistance();
}
```

Os exemplos até aqui usaram o utilitário padrão de correção ortográfica do Lucene. Mas há um outro que está em estágio experimental, que é o `DirectSpellChecker`. Ele não precisa alterar o índice original, porque faz acesso direto nos campos indicados. Essa é uma vantagem, pois não precisa do passo inicial de indexação do dicionário de referência, por outro lado ainda está com status de experimental.

```
public void testSugereComAcessoDireto() throws IOException {
    System.out
            .println("\nSugere alternativas acesso direto");
    DirectSpellChecker corretor = new DirectSpellChecker();
    Term termo = new Term(campo, termoComErro);
    int quantidadeSugestoes = 50;
    SuggestWord[] sugestoes = corretor.suggestSimilar(termo,
            quantidadeSugestoes, reader);
    for (SuggestWord sw : sugestoes) {
        System.out.println(sw.string);
    }
}
```

9.5 Sugestão de resultados (suggester)

O Lucene tem uma lista de utilitários para sugestão de complementos em um texto digitado. Esta é uma facilidade para agilizar o processo de consulta, pois permite a utilização do recurso de autocompletar em um campo de formulário. Neste caso, o usuário começa a digitar e a aplicação sugere palavras para complementar a consulta, geralmente em uma lista de acesso rápido.

O recurso de autocompletar (https://googleblog.blogspot.com.br/2008/08/at-loss-for-words.html/) com sugestões foi popularizado (não inventado) pelo Google em 2008 e explica o funcionamento do mecanismo do *Google Suggest*, que estava entrando em produção. Para quem não lembra, a funcionalidade está mostrada na figura:

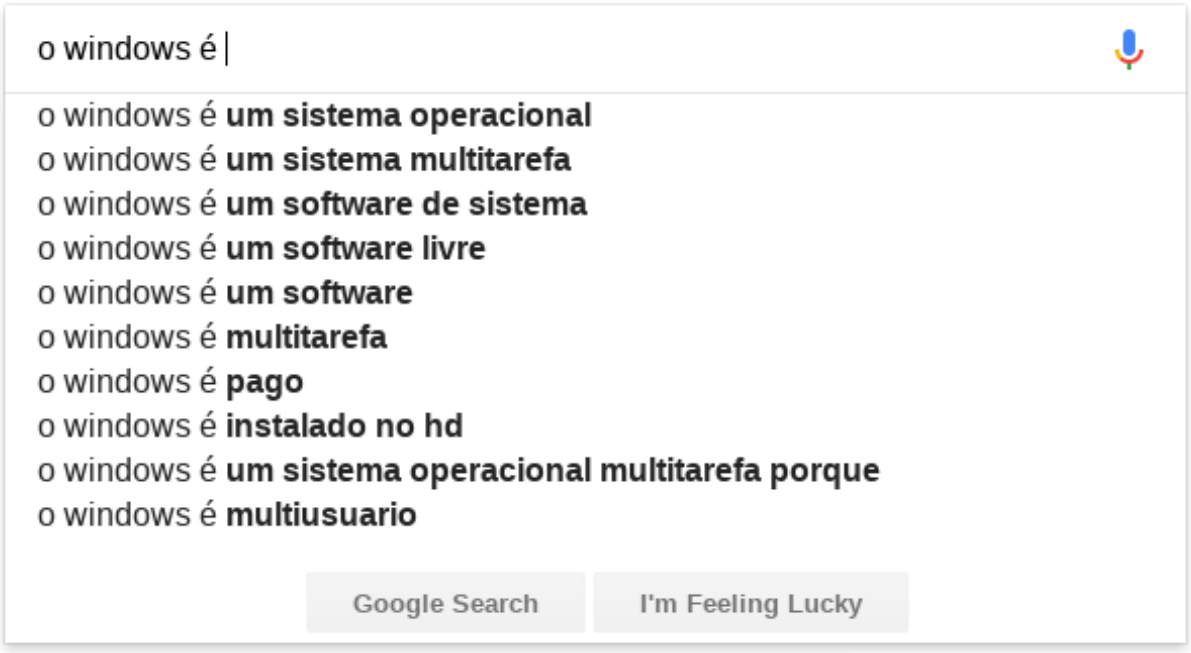


Figura 9.1: Google Suggest em ação.

Nosso exemplo usa o nome do produto para ilustrar o funcionamento do autocompletar. O usuário escreve uma parte do

nome do produto e o sistema vai mostrar uma lista de produtos que têm aquele nome parcial. A figura mostra como funciona o autocompletar com o PrimeFaces:

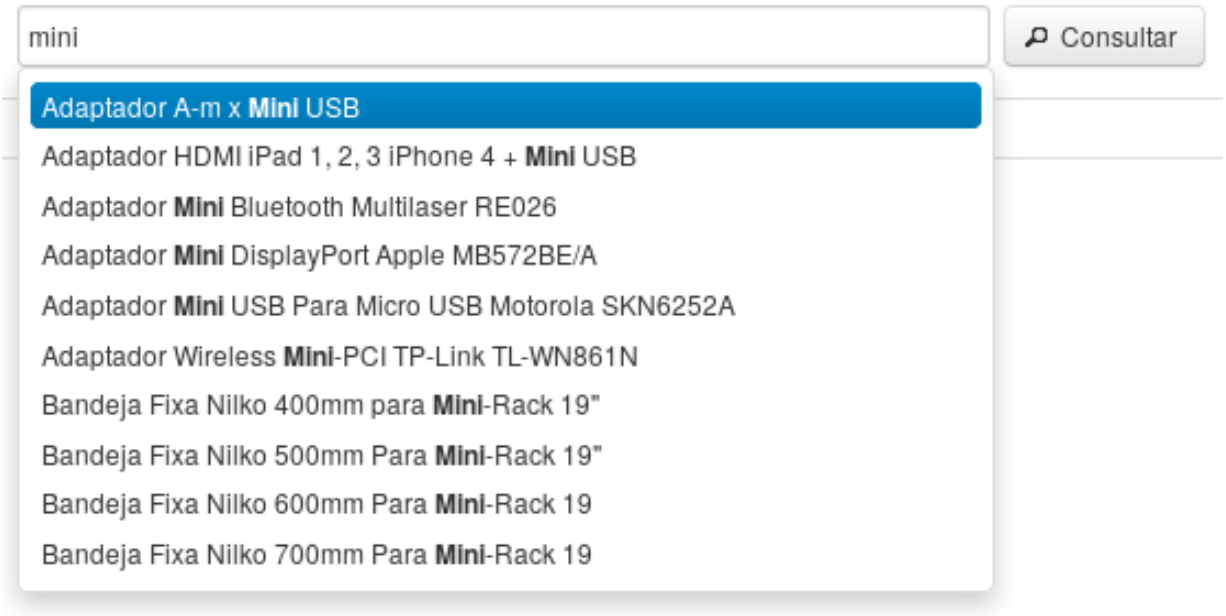


Figura 9.2: Autocompletar do PrimeFaces com Lucene.

Para acessar este recurso vamos adicionar o campo `produtoNomeNA`, no qual a sigla "NA" significa "não analisado", logo, o conteúdo deste campo não deve ser analisado pelo Lucene, o que é possível com um `StringField`. O conteúdo de um campo `StringField` é indexado como um único termo não analisado. O próximo passo é adicionar o código a seguir na classe `IndexadorProduto.preencherDadosProduto`:

```
    doc.add(new StringField("produtoNomeNA",
            produto.getNome(), Store.YES));
```

Como é um novo campo, devemos refazer o índice de produto com o `IndexadorProdutoTest`. Depois de adicionar o campo `produtoNomeNA` vamos criar o índice de sugestões para o nome dos produtos. As classes usadas neste teste são `Dictionary`, `Directory` e `AnalyzingInfixSuggester`. Com o `Dictionary` criamos um dicionário com os termos do campo `produtoNomeNA`, de forma que cada nome

completo de produto será uma sugestão. Se o `produtoNomeNA` fosse um campo analisado essa estratégia não funcionaria, porque cada palavra do nome seria uma sugestão individual.

O `Directory` para as sugestões de produtos deve ser diferente daquele usado para o índice do produto. E é bem simples o motivo. Se você usar o mesmo diretório para os dois índices, o *suggester* simplesmente vai excluir um deles. Por fim, e o mais importante, o `AnalyzingInfixSuggester` é o utilitário que cria o índice das sugestões. Para ficar claro: o índice de produto e o índice para sugestões de produtos são diferentes!

Por este motivo, a *enum* `TipoIndice` tem mais um item, o `PRODUTO_AUTOCOMPLETAR`, que aponta para o diretório `indice-produto-autocompletar`. Os testes estão em `AnalisadorSugestaoProdutoTest`. Veja o primeiro bloco a seguir, onde criamos o índice de sugestões.

```
public void test1CriaIndiceProdutoAutoCompletar()
        throws IOException {
    Dictionary dictionary =
            new LuceneDictionary(reader, "produtoNomeNA");
    Directory diretorioInfix = FSDirectory.open(Paths.get(
            TipoIndice.PRODUTO_AUTOCOMPLETAR.diretorio()));
    AnalyzingInfixSuggester analisadorSugestao =
            new AnalyzingInfixSuggester(diretorioInfix,
                    analyzer);
    analisadorSugestao.build(dictionary);
    analisadorSugestao.close();
}
```

Com o campo `produtoNomeNA` indexado, você pode listar as sugestões de nomes de produtos com o método `lookup`. Ele tem 3 parâmetros: (1) o nome parcial do produto; (2) se é para mostrar apenas os itens mais populares, opção que não está implementada nesta versão do *suggester* que é experimental; e a quantidade de sugestões para retorno. É o que vemos no método `testAnalisadorSugestaoInfix`:

```
Directory diretorioInfix = FSDirectory.open(Paths.get(
        TipoIndice.PRODUTO_AUTOCOMPLETAR.diretorio()));
```

```
AnalyzingInfixSuggester analisadorSugestao =
        new AnalyzingInfixSuggester(diretorioInfix,
                analyzer);
int quantidadeSugestoes = 100;
boolean mostrarItensPopulares = true;
List<Lookup.LookupResult> lookupResultList =
        analisadorSugestao.lookup(texto,
                mostrarItensPopulares,
                quantidadeSugestoes);
for (Lookup.LookupResult result : lookupResultList) {
    System.out.println(result.key);
}
analisadorSugestao.close();
```

Considerando uma consulta por `mini`, o resultado mostra os itens `Adaptador Mini USB`, `Adaptador Wireless Mini-PCI`, `Bandeja Fixa para Mini-Rack 19` etc. Estes itens foram consultados no índice de sugestões de produtos e não refletem as atualizações feitas no índice original de produtos. Para atualizar as sugestões quando um novo produto é incluído, usamos o método `AnalyzingInfixSuggester.add`. Da forma como está descrito no método `testAdicionaNovaSugestao`:

```
Directory diretorioInfix = FSDirectory.open(Paths.get(
        TipoIndice.PRODUTO_AUTOCOMPLETAR.diretorio()));
AnalyzingInfixSuggester analisadorSugestao =
        new AnalyzingInfixSuggester(diretorioInfix,
                analyzer);
BytesRef novaSugestao =
        new BytesRef("Adaptador mini HDMI iPad "
                + "Android Windows Linux MacOS"
                        .getBytes());
long weight = 1;
BytesRef payload = null;
analisadorSugestao.add(novaSugestao, null, weight,
        payload);
analisadorSugestao.refresh();
analisadorSugestao.close();
```

Após a execução, temos uma nova sugestão adicionada no respectivo índice. Para conferir a quantidade de itens no índice de

sugestões temos o método `getCount()`, e para verificar a contagem de memória temos o `ramBytesUsed()`, como neste trecho de código:

```
AnalyzingInfixSuggester analisadorSugestao =
        new AnalyzingInfixSuggester(diretorioInfix,
                analyzer);
System.out.println(
        String.format("Itens [%s]. Memória [%s]",
                analisadorSugestao.getCount(),
                analisadorSugestao.ramBytesUsed()));
analisadorSugestao.close();
```

O resultado desta execução mostra a quantidade de sugestões no índice e a quantidade de memória em *bytes* que está sendo usada pelo *suggester*.

```
Itens [16607]. Memória [18737]
```

Além do `AnalyzingInfixSuggester`, o Lucene dispõe de outros utilitários para sugestão, como o `AnalyzingSuggester` e o `FuzzySuggester`. O `AnalyzingSuggester` foi o primeiro deste tipo no Lucene e tem a limitação de gerar sugestões a partir de prefixos, ou seja, as sugestões são geradas a partir da primeira palavra das sugestões disponíveis e não considera as palavras intermediárias ou finais. Dessa forma, o usuário precisa saber quais as primeiras palavras da consulta, da mesma forma que o Google Suggest. Na figura do Google Suggest vista no começo da seção, o usuário digitou as primeiras palavras da consulta, no caso, "o windows é", e o buscador completou com algumas alternativas.

O `AnalyzingSuggest` é uma versão anterior e menos eficiente de gerador de sugestões. O método `testAnalisadorSugestao` mostra um exemplo de como usá-lo. Os *suggesters* são baseados na classe `Lookup`, então, a implementação dos exemplos é bem parecida. Veja:

```
Dictionary dictionary =
        new LuceneDictionary(reader, "produtoNomeNA");
String dirTemporario = System.getProperty("user.home")
```

```
        + "/livro-lucene/indice-produto-sugestao-temp";
AnalyzingSuggester analisadorSugestao =
        new AnalyzingSuggester(diretorio, dirTemporario,
                analyzer);
analisadorSugestao.build(dictionary);
int quantidadeSugestoes = 100;
boolean mostrarItensPopulares = false;
List<Lookup.LookupResult> lookupResultList =
        analisadorSugestao.lookup(texto,
                mostrarItensPopulares,
                quantidadeSugestoes);
for (Lookup.LookupResult lookupResult : lookupResultList) {
    System.out.println(lookupResult.key);
}
```

O último utilitário visto no livro é o `FuzzySuggester`. Como o nome sugere (*fuzzy* significa impreciso), ele é baseado em busca imprecisa, aquela na qual o resultado considera palavras com a grafia levemente diferente da fornecida. O exemplo da busca por `mini`, retornaria produtos como `Mindjack PS3`, `Midnight Club PS3` e `Minecraft Xbox 360`, além dos produtos com a palavra `mini`.

9.6 Consulta "More Like This" (MLT)

O utilitário `MoreLikeThis` é usado para gerar consultas baseadas em similaridade, de forma que o resultado apresente itens semelhantes. É diferente da busca tradicional por palavra-chave, onde o usuário informa geralmente entre 1 e 3 termos, e procura os textos que contêm aquelas palavras-chave específicas.

A consulta *More Like This* (MLT) é diferente porque ela usa como entrada um texto inteiro fornecido pelo usuário. A partir do conjunto de palavras deste texto, são extraídas as mais significativas para compor a busca MLT. A estratégia usada pelo Lucene é descartar termos com pouca relevância e usar as palavras mais significativas.

Esta técnica é repetida em várias funcionalidades diferentes na área de *Information Retrieval*.

A pergunta é: como saber quais as palavras significativas? E a resposta não é animadora: depende. Não existe um algoritmo perfeito e inteligente o suficiente para responder automaticamente o que é relevante ou não. E ainda tem mais um complicador, porque um termo pode ser relevante por um período determinado de tempo ou para um público específico. Quer dizer, o mesmo termo pode ser importante e não importante ao mesmo tempo, dependendo de qual é o perfil do usuário. O importante é entender que um texto extenso tem geralmente apenas poucas palavras relevantes, isto é, poucas palavras são necessárias para caracterizar aquele documento.

Exemplo 1: dado um texto sobre política, a consulta MLT retornará outros textos sobre o mesmo assunto (política) e não sobre futebol. Neste contexto, é relativamente simples resolver o problema, porque os termos comuns em um texto de política são bastante diferentes do conteúdo sobre esporte.

Exemplo 2: em um site de notícias, o sistema mostra no final da página, logo após o conteúdo principal, uma lista de notícias similares ao que o usuário está lendo naquele momento. Seria algo como "mais como isto", em uma tradução para o inglês, *more like this*. Em um e-commerce o princípio é o mesmo, enquanto o usuário vê a página de um produto, ao lado devem ser mostrados produtos similares.

A consulta MLT tem sua configuração padrão para eliminar palavras com menor relevância baseando-se no TF-IDF e BM25. Como discutido no capítulo *2. Conceitos de recuperação da informação_*, são medidas de ponderação que funcionam para a maioria dos casos com um bom nível de precisão.

Mas sempre é possível melhorar. Vamos modificar a configuração inicial do MLT para encontrar resultados diferentes. Usaremos uma heurística para encontrar a melhor configuração. Heurística é um

termo comum neste tipo de sistema, por isto estamos colocando aqui. O método heurístico é usado quando aplicamos diferentes valores para encontrar uma solução para determinado problema. Funciona na maioria das vezes, mas não há garantia de que será a melhor solução.

As configurações que serão modificadas fazem parte da classe `MoreLikeThis` e selecionam os termos relevantes com base em critérios de filtragem simples. Por exemplo, a quantidade mínima de ocorrências do termo, a quantidade mínima de documentos em que o termo aparece e o tamanho mínimo de caracteres. Nesses casos, provavelmente queremos termos que aparecem mais de uma vez no texto, que apareçam em vários documentos do índice e que tenham um tamanho mínimo, digamos, de 5 caracteres, de forma a eliminar palavras curtas.

A tabela a seguir mostra a lista de atributos relevantes para a consulta MLT:

Método	Descrição
`setBoost(boolean)`	Define se os termos devem usar o fator de elevação baseado no *ranking*. O valor padrão é *false*.
`setBoostFactor(float)`	Define o novo valor para o *boost*. O padrão é 1.
`setFieldNames(String[])`	Define os campos usados para criar a MLT durante o *runtime*.
`setMinTermFreq(int)`	Quantidade mínima de ocorrências do termo. Termos que aparecem em quantidade menor que esta serão ignorados.

Método	Descrição
`setMinDocFreq(int)`	Quantidade mínima de documentos em que o termo é citado. Se o termo aparece em quantidade menor de documentos que esta, será ignorado.
`setMaxDocFreq(int)`	Quantidade máxima de documentos em que o termo aparece.
`setMaxDocFreqPct(int)`	Percentual máximo de documentos onde o termo deve aparecer. Funciona como o `setMaxDocFreq`, entretanto, considera o percentual de documentos e não a quantidade. Termos que aparecem em um percentual maior que este serão ignorados.
`setMinWordLen(int)`	Tamanho mínimo de caracteres do termo para ser considerado na MLT.
`setMaxWordLen(int)`	Tamanho máximo do termo para ser considerado.
`setMaxQueryTerms(int)`	Quantidade máxima de termos na consulta. O padrão é 25.
`setStopWord(Set)`	Lista opcional de *stop words*, caso pretenda remover alguma palavra que considere sem relevância para a consulta.

Observação: se o campo não foi indexado com suporte a `TermVector`, você pode definir a quantidade de itens analisados através do `setMaxNumTokensParsed(int)`, que por padrão é 5000, um número um pouco grande. Pode ser útil em documentos com muito mais de 5000 palavras.

O texto usado como base da MLT no nosso exemplo será a descrição do jogo Angry Birds Star Wars:

> Prepare-se para se unir a uma aventura épica de proporções cósmicas! O universo de Angry Birds finalmente encontra o do lendário Star Wars, agora remasterizado em alta definição para os consoles, trazendo um belíssimo espetáculo visual e novas fases bônus exclusivas! Então saque o seu sabre de luz, concentre sua Força e encare os diversos desafios para salvar a galáxia em Angry Birds Star Wars!

Internamente, a MLT seleciona as palavras mais relevantes deste texto a partir de critérios estatísticos. Com esses termos é gerada uma *query* comum do Lucene que servirá para encontrar itens semelhantes. É apenas uma jogada de estatística. A partir deste texto-base construímos a consulta MLT e encontramos produtos similares, então devem aparecer no resultado produtos com alguma semelhança com aquela descrição. Daqui a pouco veremos que a consulta será feita a partir dos termos `par star wars birds angry`, extraídos do texto-base.

> **OBSERVAÇÃO:** aplicando uma heurística, podemos reduzir o tamanho do texto-base para apenas algumas poucas palavras, eliminando ruídos ou termos que trazem itens indesejados. O texto-base acima tem 65 palavras, enquanto o padrão da MLT é usar uma consulta com até 25 palavras, assim, a maioria dos termos será descartada. Com a heurística podemos reduzir o texto-base para apenas os termos mais relevantes, como visto mais à frente.

A sintaxe mais simples para usar a MLT inclui a criação da própria consulta `MoreLikeThis`, a definição do analisador e a lista dos campos utilizados. A partir daí, o procedimento é igual ao da

consulta tradicional. Estes exemplos estão na classe `ConsultaMoreLikeThisTest`. Veja:

```
    public void encontraSimilares() throws IOException {
        MoreLikeThis mlt = new MoreLikeThis(reader);
        mlt.setAnalyzer(analyzer);
        String campoProdutoDescricao = "produtoDescricao";
        mlt.setFieldNames(
                new String[] { campoProdutoDescricao });
        Reader reader = new StringReader(textoBase);
        Query query = mlt.like(campoProdutoDescricao, reader);
        TopDocs topDocs = searcher.search(query, 20);
        System.out.println(mlt.describeParams());
        logger.info(
                "Itens similares (" + topDocs.totalHits + "):");
        for (ScoreDoc sd : topDocs.scoreDocs) {
            Document doc = searcher.doc(sd.doc);
            String produtoNome = doc.get("produtoNome");
            System.out.println(produtoNome);
        }
        imprimeTermosDeInteresse(mlt, campoProdutoDescricao);
    }
```

O método `imprimeTermosPesquisados` mostra quais os termos relevantes que foram utilizados para construir a consulta MLT. No nosso exemplo, os termos são `par star wars birds angry`. O Lucene verifica tanto o texto-base quanto o índice para chegar a esses valores, como recuperado pelo `retrieveInterestingTerms`. Os ajustes finos serão explorados na próxima seção.

```
    private void imprimeTermosDeInteresse(MoreLikeThis mlt,
            String campoProdutoDescricao) throws IOException {
        Reader reader = new StringReader(textoBase);
        String[] arrTermosInteressantes =
                mlt.retrieveInterestingTerms(reader,
                        campoProdutoDescricao);
        String termosInteressantes =
                String.join(" ", arrTermosInteressantes);
        System.out.println(
                "Termos de interesse: " + termosInteressantes);
```

```
        reader.close();
    }
```

O resultado da execução é parecido com a listagem a seguir e mostra uma lista de 8191 itens similares ao texto-base, ou seja, são 8191 itens cuja descrição é parecida com o texto-base, considerando o índice de produto. Na listagem aparecem apenas os nomes dos produtos:

Angry Birds Star Wars PS3	Angry Birds Star Wars Nintendo Wii U	Angry Birds Star Wars Nintendo 3DS
Angry Birds Star Wars para PS Vita	Angry Birds Star Wars Nintendo Wii	Angry Birds Trilogy PS3
Angry Birds Trilogy Nintendo Wii U	Pen Drive Emtec 8GB EKMMD8GA102R	Pen Drive Emtec 8GB EKMMD8GA106R
LEGO Indiana Jones: The Original Adventures PS3	Lego Batman PS3	LEGO Star Wars III: The Clone Wars PS3
LEGO Star Wars: The Complete Saga PS3	LEGO Star Wars III Xbox 360	Xbox 360 Kinect Star Wars Bundle (Ed. Limitada)
Star Wars Force Unleashed PS3	Lego Indiana Jones:Original Adv + Kung Fu Panda Xbox 360	Dock Station Gear4 PG552G Angry Birds Dock Station Black Bird
Fita Star SP 300	Lego Batman Xbox 360	

Para chegar a esse resultado o Lucene considerou o texto-base e o campo `produtoDescricao`. Os detalhes desta operação serão vistos mais tarde nesta seção. Agora, vejamos o método `mlt.describeParams()`, que descreve os parâmetros usados para este exemplo inicial e fazem parte da lista de atributos relevantes vista anteriormente nesta seção:

```
maxQueryTerms  : 25
minWordLen     : 0
maxWordLen     : 0
fieldNames     : produtoDescricao
boost          : false
minTermFreq    : 2
minDocFreq     : 5
```

Personalizando a consulta MLT

Os valores iniciais dos atributos da MLT podem não ser ideais. O `maxQueryTerms` padrão é 25, um valor alto porque a consulta pode ter até 25 palavras e, com isso, perder precisão. Outro parâmetro interessante é o tamanho mínimo e máximo das palavras componentes da busca, que podem ser definidos com 5 e 15, isto é, a consulta só considera palavras com 5 a 15 caracteres, eliminando palavras muito pequenas ou grandes. Além disso, estamos usando apenas o campo `produtoDescricao`, quando uma boa opção seria usar também o `produtoNome`, que contém mais descrições sobre o produto.

Podemos mudar esses valores para aumentar ou diminuir o tamanho do resultado. Se o resultado não se mostra adequado para seus dados, faça ajustes em cada parâmetro e avalie o resultado até encontrar a configuração ideal. No código a seguir estão listadas as alterações. Exemplo: o `minDocFreq`, que por padrão é 5, foi alterado para 10. Assim, a consulta vai considerar apenas os termos que aparecem em mais de 10 documentos. O método de teste é o `encontraSimilaresComAjustes`:

```
mlt.setMinDocFreq(10);
mlt.setMinTermFreq(1);
mlt.setMinWordLen(5);
mlt.setMaxWordLen(15);
mlt.setMaxQueryTerms(15);
mlt.setBoost(true);
mlt.setBoostFactor(5);
```

Antes usamos apenas o campo `produtoDescricao`. Aqui adicionamos o campo `produtoNome`, assim, desta vez o MLT vai considerar os dois campos durante a consulta:

```
String campoProdutoDescricao = "produtoDescricao";
String campoProdutoNome = "produtoNome";
mlt.setFieldNames(new String[] { campoProdutoDescricao,
        campoProdutoNome });
```

Os parâmetros desta nova versão estão descritos a seguir:

```
maxQueryTerms  : 15
minWordLen     : 5
maxWordLen     : 15
fieldNames     : produtoDescricao, produtoNome
boost          : true
minTermFreq    : 1
minDocFreq     : 10
```

Neste caso, que é menos restritivo, a consulta encontrou 588 itens similares, enquanto a primeira versão da MLT retornou 8191 itens. É importante notar que não há muito rigor científico nesta seleção de valores com nossa heurística, o objetivo é apenas mostrar as opções e o comportamento da aplicação. Com essa configuração, os termos de interesse são `exclus aventur desafi visual encontr final univers prepar remasteriz encar bonus concentr espetacul birds angry`.

Para um sistema que necessita de mais precisão são necessárias outras ferramentas e técnicas mais complexas, que estão descritas no próximo capítulo. A partir daqui estamos indo para uma fase de experimentação, onde os resultados dependem de pesquisas e

análises diferentes para cada base de dados, uma vez que não há fórmula ou solução perfeita que atenda a todas as situações.

Consulta MLT a partir de um documento

O Lucene permite que você crie uma consulta MLT a partir de um texto-base e também através de um documento específico no índice. É claro que internamente as duas formas são parecidas, porque a consulta é construída com base nos termos de interesse extraídos do conteúdo. Mas esta consulta a partir de um documento é conveniente quando você sabe qual item o usuário selecionou.

São duas situações diferentes. Na primeira, você tem um texto-base, digamos, de uma página web, e a partir desta descrição nós buscamos no índice os itens que são similares. Neste segundo caso, você sabe o identificador do documento e a partir dele é que são selecionados os itens semelhantes.

Nos dois casos precisamos indicar qual o campo, ou campos, que usamos para encontrar os itens similares. Podemos usar tanto o campo `produtoDescricaoVetor`, aquele que foi indexado com `TermVector`, quanto o campo `produtoDescricao`. Nas novas versões, não é obrigatório o uso do `TermVector`. Entretanto, a precisão tende a ser um pouco melhor usando campos com `TermVector`, além de ser um pouco mais rápido. Isso se dá porque o `TermVector` armazena detalhes dos valores no próprio índice e por isso não é necessário fazer cálculos adicionais.

Nosso teste será feito no método a seguir, o `buscarAPartirDeUmDocumento`, onde vamos definir as configurações iniciais para filtrar termos indesejados. Esses filtros opcionais podem ser mais ou menos restritivos e foram vistos na lista de atributos relevantes da MLT.

Para usar esta funcionalidade precisamos escolher um documento e descobrir seu ID. O nosso exemplo vai usar o produto com nome `Fifa 14 Xbox One`. O bloco que encontra exatamente esse produto é:

```
    public void buscaAPartirDeUmDocumento()
            throws ParseException, IOException {
        // Encontra o identificador do documento
        int documentoID = 0;
        QueryParser parser = new QueryParser("", analyzer);
        String nomeProduto = "produtoNome:(Fifa 14 Xbox One)";
        Query queryOrigem = parser.parse(nomeProduto);
        TopDocs topdocsOrigem = searcher.search(queryOrigem, 1);
        if (topdocsOrigem.totalHits == 0) {
            throw new RuntimeException(
                    "Não encontrou nenhum documento.");
        }
        for (ScoreDoc sd : topdocsOrigem.scoreDocs) {
            Document doc = searcher.doc(sd.doc);
            documentoID = sd.doc;
            logger.info(
                    "Documento base: " + doc.get("produtoNome")
                            + " [" + documentoID + "]");
        }
        // {...}
    }
}
```

A descrição original do jogo é:

> FIFA 14 chegou para marcar a nova era da franquia! A realidade e a imersão estão à flor da pele, permitindo que os jogadores entrem de cabeça no mundo do futebol. Melhorias na jogabilidade agora permitem criar estratégias e mudar o rumo de uma partida inteira. O novo recurso, Pure Shot, faz com que cada tentativa de gol se torne uma emoção sem igual. E ainda há recursos online que incrementam ainda mais a experiência, conectando os jogadores e fãs do esporte na EA SPORTS Footbal Club. É uma verdadeira rede social do futebol, onde jogadores podem interagir, competir e compartilhar suas jogadas com milhares de pessoas do mundo todo.

A consulta MLT a partir de um documento está listada logo a seguir. Veja que a sintaxe é parecida. O método é o `like(int)` que recebe o identificador do documento como parâmetro. O restante do código é igual aos exemplos anteriores.

```
MoreLikeThis mlt = new MoreLikeThis(reader);
mlt.setMinDocFreq(10);
mlt.setMinTermFreq(1);
mlt.setMinWordLen(5);
mlt.setMaxWordLen(15);
mlt.setMaxQueryTerms(15);
mlt.setBoost(true);
mlt.setBoostFactor(5);
mlt.setStopWords(getStopWords());
Query query = mlt.like(documentoID);
TopDocs topDocs = searcher.search(query, 10);
System.out.println("Documentos similares ("
        + topDocs.totalHits + "):");
for (ScoreDoc sd : topDocs.scoreDocs) {
    Document doc = searcher.doc(sd.doc);
    System.out.println(doc.get("produtoNome"));
}
```

Os termos de interesse da *query* são `esport realidad inteir estrateg emoca sports imersa igual increment compet interag permit recurs futebol jogador` . Veja que os termos *igual* e *verdadeir* nao identificam um jogo, por isso vamos criar uma lista de *stop words* para eliminá-las. O método `getStopWords` está listado adiante e é usado para remover esses termos.

Você precisa testar algumas vezes até encontrar uma lista de *stop words* adequada à sua situação. Perceba que são termos que não ajudam a identificar o tema de um texto e geram ruído no resultado, ou seja, podem trazer itens indesejáveis. As *stop words* são importantes e merecem atenção por este motivo.

```
private Set<String> getStopWords() {
    Set<String> lista = new HashSet<>();
    lista.add("igual");
    lista.add("verdadeir");
```

```
        return lista;
    }
```

Os termos de interesse sem as *stop words* são `jogabil esport realidad inteir estrateg emoca sports imersa compet increment interag permit recurs futebol jogador` . A partir daí serão retornados os produtos similares, que vemos na próxima tabela. Veja que a eficiência da MLT é bastante razoável:

Fifa 14 PS4	Fifa 14 Xbox One	Fifa 14 PS3
Fifa 14 Português Xbox One	Fifa 14 Xbox 360	Fifa 13 Portugês Xbox 360
Espn Sports Connection Nintendo Wii U	Big League Sports Xbox 360	Kit Sports Wii Controller Gamer 6610
Sony Playstation 4 500GB	Kinect Sports Xbox 360	Fantastic Pets Xbox 360
Kinectimals Xbox 360	Fifa Street 3 PS3	Sports Champions 2 + PlayStation Move bundle PS3
Grid Racedriver PC	Grid Racedriver PS3	Controle Sony DualShock 4
Fifa 14 Nacional Xbox 360	Fifa 14 para PS Vita	

MLT sem TermVector

Há ainda mais uma opção que é usar um campo sem *TermVector*. Neste caso, precisamos definir o analisador para o Lucene encontrar os termos de interesse. São apenas 2 linhas diferentes.

Agora, usamos apenas o campo `produtoDescricao` e removemos a referência para `produtoDescricaoVetor`:

```
mlt.setAnalyzer(analyzer);
mlt.setFieldNames(new String[] { "produtoDescricao" });
```

9.7 Marcando texto com Highlighter

A marcação de texto (*highlight*) facilita a visualização dos termos da consulta dentro do texto de destino. Funciona como aquelas canetas marca-texto que usamos no papel. O objetivo é destacar apenas uma parte do texto, de forma a encontrar facilmente aquele trecho.

Como nossos sistemas são baseados em páginas web, a forma mais simples de destacar o texto é colocando as palavras selecionadas em negrito. Mas é possível personalizar para usar qualquer tipo de marcação. O Lucene implementa essa funcionalidade através da dependência *Highlighter*, indicada no bloco a seguir. Não faz parte do núcleo do Lucene, sendo implementado como um pacote externo.

```
<dependency>
    <groupId>org.apache.lucene</groupId>
    <artifactId>lucene-highlighter</artifactId>
    <version>${lucene.version}</version>
</dependency>
```

Para mostrar seu funcionamento foi criada a classe `MarcadorTextoTest`. Vamos pesquisar por *jogo do mario e luigi para wii* no índice de produtos, então, marcaremos essas palavras na descrição dos itens retornados para facilitar a visualização do resultado. Se o usuário encontrar algumas dessas palavras na descrição do produto, está justificado porque aquele item foi retornado na pesquisa.

São 3 implementações diferentes de marcadores: (1) `Highlighter`, (2) `FastVectorHighlighter` e a (3) `UnifiedHighlighter`. A primeira, `Highlighter`, não tem a melhor performance, mas por outro lado funciona em qualquer situação, sem restrições.

Highlighter

O `Highlighter` foi a primeira implementação de marcação e sua performance não é a melhor, pois a cada chamada são executados os cálculos para marcar o texto. Por isso ela pode ser um pouco mais lenta. O método `testHighlighter`, visto logo a seguir, mostra como funciona. O método importante é o `Highlighter.getBestFragment`, que faz a marcação das palavras em um texto com as *tags* de negrito do HTML `<B>` e `</B>`, extraindo um ou vários fragmentos com os termos mais relevantes.

```
// Consulta tradicional
QueryParser parser = new QueryParser("", analyzer);
Query query = parser.parse(
        "produtoDescricao:(jogo do mario e luigi para wii)");
TopDocs topDocs = searcher.search(query, 100);
String campo = "produtoDescricao";
QueryScorer scorer = new QueryScorer(query, campo);
// Marcador de texto simples
Highlighter hl = new Highlighter(scorer);
for (ScoreDoc sd : topDocs.scoreDocs) {
    Document doc = searcher.doc(sd.doc);
    String textoMarcado = hl.getBestFragment(analyzer,
            campo, doc.get(campo));
    System.out.println(
            "=== Descrição original/Com marcação ===");
    System.out.println(doc.get(campo));
    System.out.println(textoMarcado);
}
```

O `getBestFragment` retorna vários fragmentos, inclusive o mostrado a seguir. Note que não é o texto completo de entrada, apenas uma parte com as palavras da consulta marcadas com as *tags* de negrito. Veremos em seguida como marcar o texto completo, mas

deve-se imaginar que, quanto mais fragmentos, mais processamento e memória são necessários. Veja:

> New Super `<B>Mario</B>` Bros U é uma nova aventura side-scrolling com `<B>Mario</B>` , `<B>Luigi</B>` , Toad e até...

O resultado em uma página web seria esse, onde de fato os termos estão em negrito, facilitando a visualização do usuário:

> New Super **MARIO** Bros U é uma nova aventura side-scrolling com **Mario**, **Luigi**, Toad e até...

FastVectorHighlighter

O `FastVectorHighlighter` é outra implementação de marcador. Este é um pouco mais recente e usa campos indexados com o *TermVector*, mais precisamente, o campo deve ser indexado com as opções de *offsets*, *positions* e *term vectors*. Existe um custo adicional de disco pois, como visto na seção *Indexando campos com vetores*, mais informações são gravadas. Por outro lado, a marcação do texto é mais rápida exatamente porque as posições de cada palavra estão gravadas e não é necessário cálculo adicional.

O método é o `testFastVectorHighlighter` , visto adiante. Para recuperar o fragmento com texto marcado, novamente é usado o `getBestFragment` .

```
    // Consulta tradicional
    QueryParser parser = new QueryParser("", analyzer);
    Query query = parser.parse("produtoDescricaoVetor:"
            + "(jogo do mario e luigi para wii)");
    TopDocs topDocs = searcher.search(query, 100);
    // Marcador de texto
    FastVectorHighlighter fhl = new FastVectorHighlighter();
    FieldQuery fq = fhl.getFieldQuery(query, reader);
    String campo = "produtoDescricaoVetor";
```

```
for (ScoreDoc sd : topDocs.scoreDocs) {
    Document doc = searcher.doc(sd.doc);
    int tamanho = doc.get(campo).length();
    String textoMarcado = fhl.getBestFragment(fq,
            reader, sd.doc, campo, tamanho);
    System.out.println(
            "=== Descrição original/Com marcação ===");
    System.out.println(doc.get(campo));
    System.out.println(textoMarcado);
}
```

O resultado da execução é levemente diferente do *highlighter* anterior porque esta implementação permite que seja informado o tamanho do fragmento. Veja:

> Super `<b>Mario</b>` Bros U é uma nova aventura side-scrolling com `<b>Mario</b>` , `<b>Luigi</b>` , Toad e até mesmo seu personagem Mii! Agora é sua chance para entrar no Reino do Cogumelo e explorar novos mundos, novos poderes e novas maneiras de `<b>jogar</b>` .

Marcando múltiplos trechos

Em textos extensos os termos da consulta podem estar separados e distantes, dificultando sua visualização. Para resolver isso, uma alternativa é usar o método `getBestFragments` , que retorna uma lista de pequenos fragmentos de texto, e não apenas um longo texto com as marcações.

O código a seguir divide o texto completo em até 3 fragmentos. Observe que o tamanho mínimo de um fragmento é 18, para esta versão do Lucene. Muito do que é visto neste capítulo é experimental e sofre mudanças frequentes. Veja:

```
int qtdFragmentos = 3;
for (ScoreDoc sd : topDocs.scoreDocs) {
    Document doc = searcher.doc(sd.doc);
    int tamanho =
```

```
                doc.get(campo).length() / qtdFragmentos;
        if (tamanho < 18) { // O tamanho mínimo do fragmento é 18
            tamanho = 18;
        }
        String[] textosMarcados =
                fhl.getBestFragments(fq, reader, sd.doc,
                        campo, tamanho, qtdFragmentos);
        System.out.println(
                "=== Descrição original/Com marcações ===");
        System.out.println(doc.get(campo));
        if (textosMarcados.length > 0) {
            for (int i = 0; i < textosMarcados.length; i++) {
                System.out.println("Frag " + i + ": "
                        + textosMarcados[i]);
            }
        } else
            System.out.println("Nenhum texto marcado");
    }
```

O resultado mostra 3 fragmentos com negrito para a descrição do jogo. Veja:

```
Frag 0: Super <b>Mario</b> Bros U é uma nova aventura side-scrolling com
<b>Mario</b>, <b>Luigi</b>, Toad
Frag 1: seu personagem Mii!  Agora é sua chance <b>para</b> entrar no Reino
do Cogumelo e explorar
Frag 2:  novos poderes e novas maneiras de <b>jogar</b>.
```

Personalizando o marcador

A marcação foi feita por meio das *tags* `<b>` e `</b>` até aqui. Para mudar esse formato podemos usar uma variação do método `getBestFragment`, adicionando os parâmetros `FragListBuilder`, `FragmentsBuilder`, `preTags`, `postTags` e `Encoder`. Então mudamos as *tags* para `<bold>` e `</bold>`, visto neste exemplo completo:

```
    FragListBuilder fragListBuilder =
            new SimpleFragListBuilder();
```

```
FragmentsBuilder fragBuilder =
        new ScoreOrderFragmentsBuilder();
String[] preTags = new String[] { "<bold>" };
String[] postTags = new String[] { "</bold>" };
Encoder encoder = new DefaultEncoder();
String campo = "produtoDescricaoVetor";
for (ScoreDoc sd : topDocs.scoreDocs) {
    Document doc = searcher.doc(sd.doc);
    int tamanho = doc.get(campo).length();
    String textoMarcado = fhl.getBestFragment(fq, reader,
            sd.doc, campo, tamanho, fragListBuilder,
            fragBuilder, preTags, postTags, encoder);
    System.out.println(
            "=== Descrição original/Com marcação ===");
    System.out.println(doc.get(campo));
    System.out.println(textoMarcado);
}
```

O resultado esperado é que os termos da consulta fiquem entre as novas *tags* `<bold>` e `</bold>`, assim:

```
Super <bold>Mario</bold> Bros U é uma nova aventura side-scrolling com
<bold>Mario</bold>, <bold>Luigi</bold>, Toad e até mesmo seu personagem
Mii!  Agora é sua chance <bold>para</bold> entrar no Reino do Cogumelo e
explorar novos mundos, novos poderes e novas maneiras de
<bold>jogar</bold>.
```

UnifiedHighlighter

O marcador mais recente é o `UnifiedHighlighter`, que apresenta boas opções de personalização e performance, mas ainda está em fase experimental. Ele pode ser usado em campos indexados com ou sem as posições do *TermVector*. Caso o campo não tenha as posições do *TermVector*, o *analyzer* processa novamente o conteúdo do campo para marcar as palavras.

A forma mais simples de usá-lo é conforme o bloco a seguir. De forma geral, é bem próximo dos demais exemplos, tanto que o código está mostrando apenas as linhas que são diferentes. A diferença neste exemplo é que a marcação é feita apenas em uma única frase, escolhida pelo próprio `UnifiedHighlighter`, enquanto o resto do texto é ignorado. Este é o código:

```
// Marcador de texto experimental
UnifiedHighlighter uhl =
        new UnifiedHighlighter(searcher, analyzer);
String[] textosMarcados =
        uhl.highlight(campo, query, topDocs);
int i = 0;
for (ScoreDoc sd : topDocs.scoreDocs) {
    Document doc = searcher.doc(sd.doc);
    System.out.println(
            "=== Descrição original/Com marcação ===");
    System.out.println(doc.get(campo));
    System.out.println(textosMarcados[i]);
    i++;
}
```

Por padrão, este marcador seleciona apenas a frase mais relevante, ignorando o resto do conteúdo. Para marcar o texto completo (e não uma única frase) redefinimos o método `UnifiedHighlighter.getBreakIterator` com a classe `WholeBreakIterator`. Dessa forma:

```
UnifiedHighlighter uhl =
        new UnifiedHighlighter(searcher, analyzer) {
            @Override
            protected BreakIterator getBreakIterator(
                    String field) {
                return new WholeBreakIterator();
            }
        };
```

Outra possibilidade é fazer o *highlight* em mais de um campo. Imagine que a consulta é feita em 2 campos, digamos, `produtoDescricao` e `produtoNome`. O *highlight* nestes campos é feito

pelo método `highlightFields`. Conforme o exemplo adiante, a consulta é diferente para cada campo, por isso, o marcador retorna um mapa com 2 conjuntos de valores, um para cada campo da consulta:

```
QueryParser parser = new QueryParser("", analyzer);
Query query = parser.parse(
        "produtoDescricao:(jogo do mario e luigi para wii) "
                + "AND produtoNome:(mario e luigi)");
TopDocs topDocs = searcher.search(query, 100);
UnifiedHighlighter uhl =
        new UnifiedHighlighter(searcher, analyzer);
String[] campos = { "produtoDescricao", "produtoNome" };
Map<String, String[]> mapaFragmentos =
        uhl.highlightFields(campos, query, topDocs);
for (String chave : mapaFragmentos.keySet()) {
    String[] fragmentos = mapaFragmentos.get(chave);
    System.out.println("====\nCampo: " + chave);
    for (String frag : fragmentos) {
        System.out.println(frag);
    }
}
```

Marcação com sinônimos

Naturalmente que o marcador tem de funcionar com os sinônimos. A alteração é bem simples, apenas vamos trocar o analisador de `BrazilianAnalyzer` para o já conhecido `AnalisadorSinonimos`, que usa o arquivo `sinonimos.txt` para guardar os termos similares. Considerando que `wii` e `nintendo` são sinônimos, a consulta `jogo do mario e luigi para wii` e `jogo do mario e luigi para nintendo` são equivalentes.

O código está no método `testHLSinonimo`. Note que usamos o `AnalisadorSinonimos` que vai processar a consulta considerando a lista de sinônimos definida. A seguir mostramos a alteração.

```
analyzer = new AnalisadorSinonimos();
QueryParser parser = new QueryParser("", analyzer);
```

```
Query query =
        parser.parse("produtoDescricao:(jogo do mario "
                + "e luigi para nintendo)");
```

9.8 Faceted search/navigation

Faceted search (busca com facetas) ou *faceted navigation* (navegação com facetas) é uma técnica usada para dividir o resultado da busca em categorias. As facetas, geralmente, mostram a quantidade de itens em cada categoria entre parênteses. Assim, os *facets* são uma forma de classificação usada para organizar itens de uma coleção por meio das categorias. É um recurso que se popularizou com o *e-commerce*, onde é visto com mais frequência.

O exemplo mais evidente é a busca por um produto nos departamentos da loja. Imagine uma busca livre por *mario e luigi* em um *e-commerce* como o Mercado Livre. Como visto na imagem a seguir, o resultado da busca mostra que há produtos em várias categorias, incluindo *games*, bonecos, casa, Lego etc. A partir da lista, o usuário navega entre as categorias para encontrar o produto que procura.

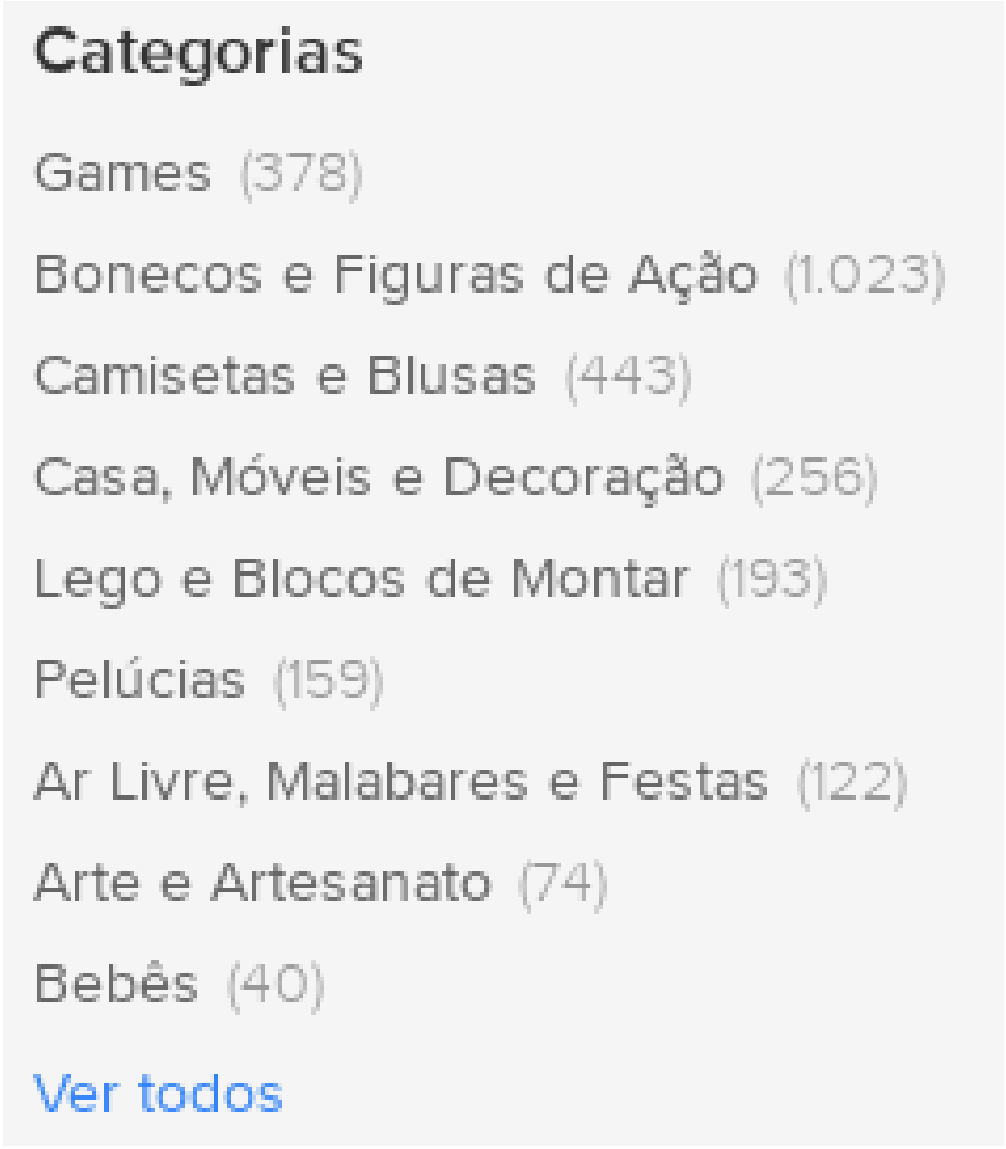


Figura 9.3: Buscando por "Mario e Luigi" no Mercado Livre.

Outro exemplo é a classificação por faixa de preços. As faixas podem ser $1 a $99, $100 a $199, $200 a $299 e a partir de $300. Quando você quer comprar um tênis entre $100 e $200, o que estiver fora desta faixa será removido da consulta. Outras opções mais avançadas serão vistas no fim da seção.

Facets com Lucene

O Lucene tem uma API específica para tratar *facets*, o `lucene-facet`. O buscador da Sematext (https://sematext.com/opensee/search/) é um bom ponto inicial para ver sua implementação na prática. Voltando ao nosso exemplo, para adicionar os *facets* ao seu projeto use a dependência:

```
<dependency>
    <groupId>org.apache.lucene</groupId>
    <artifactId>lucene-facet</artifactId>
    <version>${lucene.version}</version>
</dependency>
```

A classe `IndexadorProdutoFacet` implementa a indexação básica com *facets* para String e para intervalo de valores. Começamos pelo método `indexaDocValues()`, que cria o índice com *facets* para a categoria e preço do produto. Será criado um novo diretório para este índice, adicionado em `TipoIndice.PRODUTO_DOCVALUES`. Isso evita que o teste de um recurso específico atrapalhe os outros.

O ponto inicial dos *facets* é a classe `FacetsConfig`, onde são feitas as configurações das dimensões. Nos *facets*, os campos pertencem a uma dimensão e a contagem é feita através destas dimensões. Por exemplo, a opção `setMultiValued` indica que uma dimensão pode ou não receber mais de um valor. Neste exemplo, um produto pode pertencer a mais de uma categoria:

```
FacetsConfig config = new FacetsConfig();
config.setMultiValued("categoriaDim", true);
```

A seguir indexamos os produtos da base. Para reduzir o código indexamos apenas alguns poucos campos, no caso `produtoNome`, `produtoId`, `categoriaDim` e `preco`. As classes deste bloco são a (1) `SortedSetDocValuesFacetField` para gravar dimensões do tipo String e a (2) `DoubleDocValuesField` para valores do tipo `Double`.

Observação: para números tipo `Long` use o `NumericDocValuesField` e para `Float`, use `FloatDocValuesField`. Outra diferença é no momento de adicionar o campo no índice, o que é feito com a instrução

`writer.addDocument(config.build(doc))` para que o `FacetsConfig` crie as estruturas necessários ao funcionamento dos *facets*:

```
    for (Produto produto : produtos) {
        Document doc = new Document();
        doc.add(new StringField("produtoId",
                produto.getId().toString(), Store.YES));
        doc.add(new TextField("produtoNome",
                produto.getNome(), Store.YES));
        for (Categoria categoria : produto
                .getCategorias()) {
            doc.add(new SortedSetDocValuesFacetField(
                    "categoriaDim",
                    categoria.getNome()));
        }
        doc.add(new DoubleDocValuesField("preco",
                produto.getPreco().doubleValue()));
        writer.addDocument(config.build(doc));
    }
    writer.commit();
```

Para fechar os recursos, o exemplo usa a instrução `IOUtils.close(analyzer, writer, indexDir)`. A ideia é mostrar mais alguns recursos disponíveis no Lucene, no caso, o `IOUtils` que implementa funcionalidades para tratar o sistema de arquivos.

Nesse exemplo base dos *facets*, a classe `IndexadorProdutoDocValuesTest` implementa os testes. Para criar o índice temos o método `testIndexaFacetDocValues` que apenas faz uma chamada ao `IndexadorProdutoFacet.indexaDocValues()`. O índice com *facets*, assim como os demais, só precisa criado uma vez e durante o uso da aplicação deve ser apenas atualizado.

O método interessante mesmo é o `testBuscaDocValues` que nos apresenta a busca básica com *facets*. As novidades são as classes (1) `SortedSetDocValuesReaderState` que é um utilitário para processar as informações do *facet* e fazer os cálculos das dimensões, (2) `FacetsCollector` que coleta o resultado da busca e (3) `Facets` que armazena as quantidades de itens em cada dimensão.

Um detalhe é que o `TopDocs` não será trabalhado nestes exemplos, uma vez que eles são apenas o resultado da busca tradicional e não vamos mostrar os documentos em uma página web. Assim, o `TopDocs` vai ficar sobrando no código porque o objetivo é mostrar apenas os *facets* trabalhando.

> Para recuperar apenas os *facets* (sem recuperar os `TopDocs`) use a instrução `searcher.search(query, collector)` no lugar do `TopDocs topDocs =... .`

```
// (1)
SortedSetDocValuesReaderState state =
        new DefaultSortedSetDocValuesReaderState(
                reader);
// (2)
FacetsCollector collector = new FacetsCollector();
QueryParser parser = new QueryParser("", analyzer);
Query query =
        parser.parse("produtoNome:(mario e luigi)");
TopDocs topDocs = FacetsCollector.search(searcher,
        query, 100, collector);
// (3)
Facets facets = new SortedSetDocValuesFacetCounts(state,
        collector);
```

Com os dados coletados, é hora de mostrar os valores calculados. Para efeito de teste, vamos limitar o resultado a 10 categorias (4). O (5) `FacetResult` recupera o resultado para a dimensão `categoriaDim` , que foi indexada no começo do exemplo. A lógica do item (6) é para limitar a quantidade de itens mostrados. Por fim, a classe (7) `LabelAndValue` mostra a combinação entre o nome do *facet* e a quantidade de itens que ela contém.

```
// (4)
int qtdCategorias = 10;
// (5)
FacetResult result = facets
        .getTopChildren(qtdCategorias, "categoriaDim");
```

```
// (6)
if (result.childCount < qtdCategorias) {
    qtdCategorias = result.childCount;
}
for (int i = 0; i < qtdCategorias; i++) {
    // (7)
    LabelAndValue labelValue = result.labelValues[i];
    System.out.println(labelValue.label + " ("
            + labelValue.value + ")");
}
```

O resultado é mostrado neste bloco de código. Para a consulta inicial `produtoNome:(mario e luigi)"` temos produtos nessas categorias:

```
Jogos para Wii (13)
Consoles de Video Game (8)
Jogos para Nintendo 3DS (8)
Jogos para Wii U (6)
Acessórios para Games (2)
Controles (2)
```

Mas e quanto o usuário quer uma restrição por intervalo de preço? Os passos estão descritos a seguir. Primeiro, defina os intervalos de valores. O campo `preco` é `Double`, então, usamos um `DoubleRange`. Para campos `Long` temos o `LongRange`:

```
// (1)
DoubleRange[] intervalos = new DoubleRange[4];
intervalos[0] =
        new DoubleRange("Até $99", 0, true, 99, true);
intervalos[1] = new DoubleRange("$100-$199", 100, true,
        199, true);
intervalos[2] = new DoubleRange("$200-$299", 200, true,
        299, true);
intervalos[3] = new DoubleRange("Mais de $300", 300,
        true, Double.MAX_VALUE, true);
```

Depois, crie a mesma estrutura do *facet* anterior, com (2) `FacetsCollector`. A diferença é que o contador dos *facets* é o (3)

`DoubleRangeFacetCounts` , ou `LongRangeFacetCounts` se os valores forem `Long` .

```
    // (2)
    FacetsCollector collector = new FacetsCollector();
    QueryParser parser = new QueryParser("", analyzer);
    Query query =
            parser.parse("produtoNome:(mario e luigi)");
    TopDocs topDocs = FacetsCollector.search(searcher,
            query, 100, collector);
    // (3)
    DoubleRangeFacetCounts facets =
            new DoubleRangeFacetCounts("preco", collector,
                    intervalos);
```

E o resultado da consulta com *facets* para intervalos é mostrado a seguir. Novamente, estamos falando da consulta por *mario e luigi* e esta lista mostra a quantidade de produtos em cada faixa:

```
Até $99 (8)
$100-$199 (4)
$200-$299 (9)
Mais de $300 (18)
```

Esses dois exemplos são os casos básicos, em que uma categoria não tem subcategorias. Uma situação um pouco mais complexa é quando existe uma hierarquia de categorias, ou seja, existe uma categoria geral e subcategorias associadas. Isso cria um sistema de classificação chamado de taxonomia. Taxonomia é um termo pouco conhecido mas que faz grande diferença na construção de sistemas complexos. E é um bom recurso para melhorar a experiência do usuário. O próximo tópico explica o que vem a ser uma taxonomia.

Indexando com taxonomias

Taxonomia é uma forma de classificar objetos para indicar relacionamentos. Inicialmente, a taxonomia foi usada para classificar plantas e animais, mas é usada agora para classificar diversas áreas do conhecimento, incluindo a computação. A taxonomia é

também uma ciência que estuda a classificação, mas o Lucene não entra neste nível de detalhe, nem nós entraremos.

A taxonomia faz parte do nosso cotidiano desde a infância, quando você precisava ir à biblioteca pegar um livro. No sistema de classificação da biblioteca, cada estante é dedicada a um tema e as prateleiras guardam os subtópicos relacionados ao tema central. Essa organização segue um modelo lógico fácil de entender, partindo de uma categoria geral para categorias relacionadas cada vez mais específicas.

Para encontrar um livro sobre Lucene, por exemplo, você pode seguir a seguinte sugestão de taxonomia: *Computers / Programming / Languages / Java / Enterprise Edition / Search Engines*. O primeiro nível é *Computers*, o mais genérico. O nível 2 é *Programming*, um nível mais específico de *Computers*. O nível 3 é *Languages*, que é uma subcategoria de *Programming*. E continua seguindo até o último nível onde se encontra o Lucene, *Search Engines*. Essa taxonomia faz parte do DMOZ (http://dmoztools.net/), um projeto independente que ajuda a organizar os sites da internet.

Se não houvesse um sistema de classificação bem definido como o da biblioteca ou o DMOZ, encontrar livros seria bem mais complicado. No Brasil, o CNPq tem uma lista para classificar as áreas do conhecimento, que pode ser vista na imagem. Há níveis gerais, como Ciências Exatas e da Terra com subníveis, incluindo Matemática, Probabilidade e Computação. Veja:

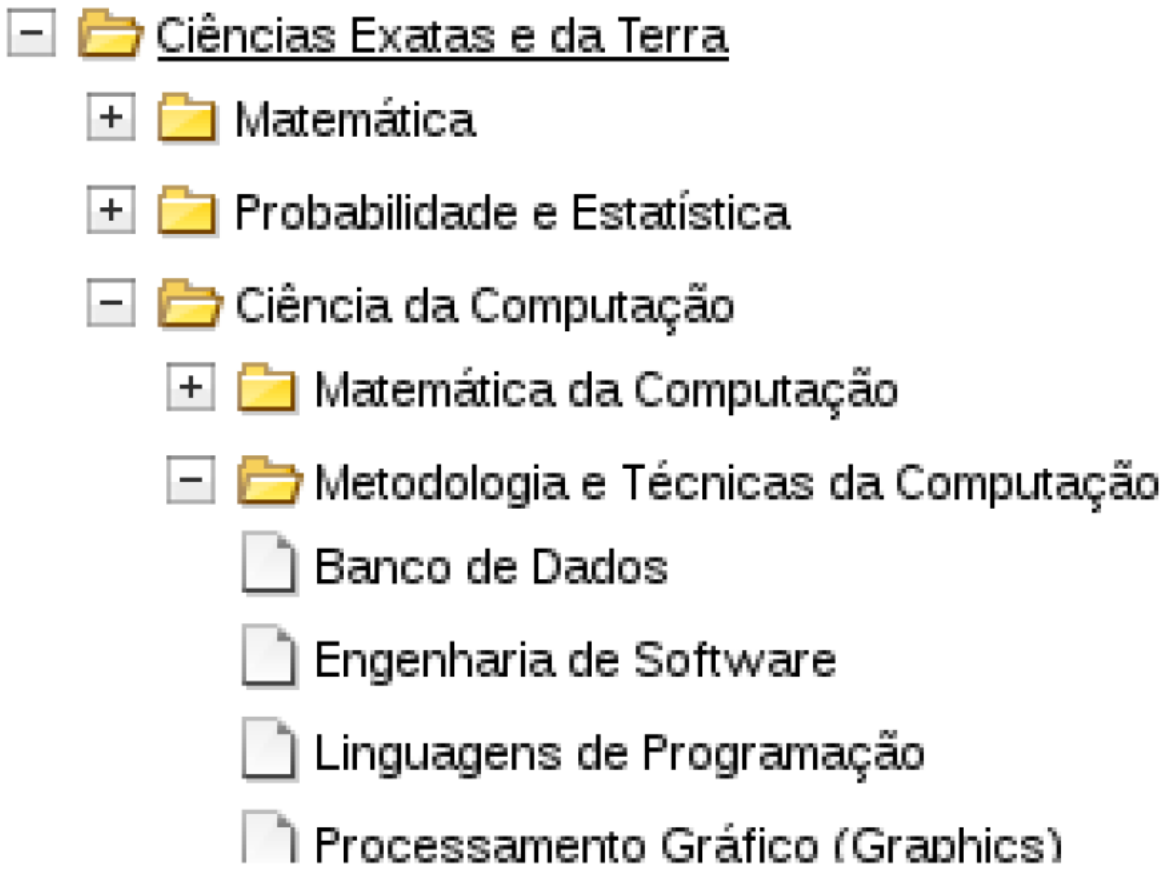


Figura 9.4: Taxonomia das áreas do conhecimento.

Em uma loja de departamentos funciona da mesma forma. Temos as categorias gerais e suas especializações. Veja essas taxonomias para uma bola e para um óculos de natação:

- Departamentos / Futebol / Bolas / Bola Futebol Campo Nike;
- Departamentos / Natação / Óculos para Natação / Óculos Speedo New Shark.

O recurso da taxonomia é somado à busca tradicional por palavra-chave para agilizar a vida do usuário durante a navegação, que pode procurar por um livro de ficção (nível 1 da taxonomia) e ação (nível 2) do Dan Brown (palavra-chave) que custe entre $5 e $15 (intervalo de valor). Com um bom sistema de taxonomias, ele encontraria facilmente este item na loja.

Existem categorias criadas com critérios claros, principalmente na área de Biologia e Direito. Mas há taxonomias mais flexíveis, digamos assim. Veja o caso da Netflix, com as categorias Ação e Aventura, Infantil, Comédia e Documentários, que são gêneros razoáveis. Agora veja outros exemplos de subcategorias: filmes baseados em livros infantis, filmes épicos, filmes sobre vinho e apreciação de bebidas (https://www.netflix.com/browse/genre/1458) (existe mesmo), dramas independentes realistas da Grã-Bretanha etc. Portanto, a taxonomia é criada para o contexto da sua aplicação e é você quem define as regras.

Para ilustrar esse conceito criaremos uma taxonomia para o nome dos produtos, onde cada palavra do nome será um nível da hierarquia. Então, para o produto *Apple iPhone 5c 32GB*, a taxonomia terá os níveis *apple / iphone / 5c / 32gb*. Para *Apple Macbook Air* a taxonomia será algo como *apple / macbook / air* e para *Apple Adaptador USB* seria *apple / adaptador / usb*.

O código para indexar com a taxonomia está no método `indexaTaxonomia()` da classe `IndexadorProdutoFacet`. Para usar a taxonomia são necessários 2 índices separados. Aqui criamos o `TipoIndice.PRODUTO_FACET` para o índice base e o `TipoIndice.PRODUTO_TAXO` para a taxonomia. Este índice lateral ou secundário é criado pelo (1) `DirectoryTaxonomyWriter`. Criamos a dimensão `produtoNomeDim` e indicamos que ela é uma hierarquia com o (2) `setHierarchical("produtoNomeDim", true)`. Significa este *facet* segue uma ordem e cada item representa um nível.

```
// (1)
DirectoryTaxonomyWriter taxoWriter =
        new DirectoryTaxonomyWriter(taxoDir);
FacetsConfig config = new FacetsConfig();
config.setMultiValued("categoriaDim", true);
// (2)
config.setHierarchical("produtoNomeDim", true);
```

Na sequência vamos preencher a taxonomia com o nome do produto. Os blocos (1) e (2) adicionam os novos *facets*. São 2 novas

dimensões na nossa aplicação. Além da taxonomia com o nome do produto, teremos uma dimensão com a letra inicial do nome do produto. É um *facet* bastante utilizado na busca por nomes.

O bloco (1) cria um *array* com cada palavra do nome do produto. Para um iPhone, seria algo como `[apple , iphone , 5c , 32gb]`. Cada item é uma palavra do nome do produto, assim, no *facet*, cada palavra será um nível na hierarquia da taxonomia. Para adicionar um campo do tipo *facet* usamos a classe `FacetField`. O trecho (2) extrai a primeira letra de cada nome e cria a dimensão também com o `FacetField`. O trecho (3) adiciona o documento no índice de produto e no índice da taxonomia. O último trecho (4) fecha o índice da taxonomia e faz as gravações no disco:

```
for (Produto produto : produtos) {
    // {...}
    // (1)
    String[] arrNome = produto.getNome().trim()
            .toLowerCase().split("\\s+");
    doc.add(new FacetField("produtoNomeDim",
            arrNome));
    // (2)
    String primeiraLetra = produto.getNome()
            .substring(0, 1).toLowerCase();
    doc.add(new FacetField("letraInicialDim",
            primeiraLetra));
    // {...}
    // (3)
    writer.addDocument(
            config.build(taxoWriter, doc));
}
// (4)
taxoWriter.commit();
taxoWriter.close();
```

O índice do produto e o da taxonomia foram criados e agora vamos navegar pelos *facets* por meio da classe `IndexadorProdutoFacetTest`. Começamos com o método `testTaxonomiaProduto` abrindo o diretório da taxonomia (1) através do `TaxonomyReader`. A consulta neste

exemplo é `produtoNome:(apple)`. A configuração para usar a dimensão `produtoNomeDim` fica no trecho (2). O trecho (3) usa o `FastTaxonomyFacetCounts` para calcular os *facets* e, por fim, em (4) os resultados são mostrados para as 5 primeiras categorias encontradas.

```
// (1)
Directory taxoDir = FSDirectory.open(
        Paths.get(TipoIndice.PRODUTO_TAXO.diretorio()));
TaxonomyReader taxoReader =
        new DirectoryTaxonomyReader(taxoDir);
FacetsCollector collector = new FacetsCollector();
QueryParser parser = new QueryParser("", analyzer);
Query query = parser.parse("produtoNome:(apple)");
TopDocs topDocs = FacetsCollector.search(searcher,
        query, 100, collector);
// (2)
FacetsConfig config = new FacetsConfig();
config.setHierarchical("produtoNomeDim", true);
// (3)
Facets taxoFacets = new FastTaxonomyFacetCounts(
        taxoReader, config, collector);
// (4)
int qtdCategorias = 5;
FacetResult result = taxoFacets
        .getTopChildren(qtdCategorias, "produtoNomeDim");
```

O resultado da execução mostra as 5 primeiras categorias:

```
apple (72)
ipad (40)
ipod (29)
case (10)
adaptador (6)
```

Vamos simular a navegação do usuário e selecionar um desses itens, digamos, a categoria *ipad*. Isso é feito passando como parâmetro o caminho na taxonomia. Desta forma:

```
FacetResult result = taxoFacets.getTopChildren(
        qtdCategorias, "produtoNomeDim", "ipad");
```

O resultado mostra as subcategorias para a categoria *ipad*. São estas as opções:

```
4 (17)
mini (14)
3 (4)
2 (4)
apple (1)
```

Vamos navegar novamente e restringir a consulta para a subcategoria *mini*:

```
    FacetResult result =
            taxoFacets.getTopChildren(qtdCategorias,
                    "produtoNomeDim", "ipad", "mini");
```

O resultado está listado a seguir e você pode repetir o processo com cada item até chegar à última categoria:

```
16gb (5)
32gb (5)
64gb (4)
```

Explorando facets com DrillDown

O *drill down* é a ação de passar de uma informação mais genérica para uma mais detalhada, geralmente aprofundando o nível em uma estrutura. É o que fazemos em uma lista de diretórios quando abrimos uma das pastas. Você faz o *drill down* para olhar dentro do diretório. Se houver outra lista de pastas, você pode fazer novamente o *drill down* e entrar em um nível mais específico.

No Lucene, existe um tipo específico de consulta chamada `DrillDownQuery` para esta situação. Ela é usada quando você quer combinar vários *facets*. No código a seguir (método `testDrillDown()`) estamos simulando uma consulta pela palavra-chave *apple*, onde os produtos começam com a letra "a" e vamos mostrar os valores da dimensão `produtoNomeDim` :

```
    // (1)
    FacetsConfig config = new FacetsConfig();
    config.setHierarchical("produtoNomeDim", true);
    // (2)
    QueryParser parser = new QueryParser("", analyzer);
    Query baseQuery = parser.parse("produtoNome:(apple)");
    // (3)
    DrillDownQuery query =
            new DrillDownQuery(config, baseQuery);
    query.add("letraInicialDim", "a");
    // (4)
    FacetsCollector collector = new FacetsCollector();
    searcher.search(query, collector);
    Facets taxoFacets = new FastTaxonomyFacetCounts(
            taxoReader, config, collector);
    FacetResult result =
            taxoFacets.getTopChildren(5, "produtoNomeDim");
```

Para esta combinação de `produtoNome:apple` com `letraInicialDim=a` o resultado mostra os produtos que contêm a palavra *apple* no nome e que começam com a letra "a":

```
apple (72)
adaptador (6)
access (1)
```

Com os *facets* você pode fazer uma análise exploratória no conteúdo do índice, adicionando vários parâmetros na `DrillDownQuery` e depois visualizando todas as dimensões existentes. Seria uma simulação da navegação natural do usuário no site, quando ele passeia livremente pelos filtros disponíveis.

No bloco (1) a seguir vamos filtrar os produtos que comecem com as letras "a" ou "i", bem como as categorias "Impressoras" ou "Jogos para PS3". No final, usamos o `getAllDims()` para mostrar as dimensões disponíveis nos campos (2). Veja o método `testDrillDownAllDim`:

```
    // (1)
    DrillDownQuery query = new DrillDownQuery(config);
```

```
    query.add("letraInicialDim", "i");
    query.add("letraInicialDim", "a");
    query.add("categoriaDim", "Impressoras");
    query.add("categoriaDim", "Jogos para PS3");
    FacetsCollector collector = new FacetsCollector();
    searcher.search(query, collector);
    Facets taxoFacets = new FastTaxonomyFacetCounts(
            taxoReader, config, collector);
    // (2)
    int qtdCategorias = 3;
    List<FacetResult> results =
            taxoFacets.getAllDims(qtdCategorias);
    System.out.println(results);
```

O resultado mostra as dimensões disponíveis. Dentro de `categoriaDim` temos 158 produtos (*value*) e apenas 2 categorias (*childCount*). A `letraInicialDim` tem os mesmos 158 produtos e também 2 categorias. A dimensão `produtoNomeDim` tem novamente 158 produtos e 20 categorias. Na listagem limitamos o número de itens para mostrar apenas os 3 primeiros:

```
[dim=categoriaDim path=[] value=158 childCount=2
  Impressoras (126)
  Jogos para PS3 (32)
, dim=letraInicialDim path=[] value=158 childCount=2
  i (135)
  a (23)
, dim=produtoNomeDim path=[] value=158 childCount=20
  impressora (123)
  assassins (9)
  impressora laser (3)
]
```

O número de produtos é sempre igual porque o filtro é aplicado a todas as dimensões. Mas existem casos especiais. Vejamos quais são.

Explorando com DrillSideways

O `DrillDown` permite a navegação entre os *facets*, que no final das contas são filtros. Quando um item é selecionado, aquele filtro é aplicado e apenas os elementos correspondentes ficam da lista, assim, com o `DrillDown` o filtro é aplicado em todas as dimensões. Entretanto, há casos especiais onde queremos aplicar o filtro em algumas dimensões e não em todas.

Considere um índice com 3 dimensões: `letraInicialDim`, `produtoNomeDim` e `categoriaDim`, usando o recurso do `DrillDown`. Quando você aplicar o filtro `letraInicialDim=a`, a única letra disponível nesta dimensão será "a". Da mesma forma, as outras dimensões só conterão produtos que iniciem com a letra "a". Os outros produtos não devem aparecer porque o filtro é exatamente para isso.

Mas e se o usuário quiser filtrar por outra letra da dimensão? Não é possível porque a única letra disponível é "a", lembra? Essa é a ideia do *drill up*, o inverso do *drill down*, que seria a ação de passar de algo mais específico e voltar para o genérico. Como o *drill down* apagou as outras letras, isso não é possível.

Usando o `DrillSideways` resolvemos essa limitação. Não é exatamente um problema, porque em muitos sites o comportamento ainda é de `DrillDown`, mas a usabilidade moderna permite o uso do `DrillSideways`, afinal, porque o usuário tem mais flexibilidade na navegação.

Esta é a função do `DrillSideways`, um tipo especial de `DrillDown` que faz a filtragem lateral dos valores. O filtro é aplicado em algumas das dimensões do índice, menos no campo do próprio filtro. Ok, é realmente um pouco complicado, mas com os exemplos ficará mais simples de entender. Este é o código do método `testDrillSideways`:

```
FacetsConfig config = new FacetsConfig();
config.setHierarchical("produtoNomeDim", true);
DrillDownQuery query = new DrillDownQuery(config);
query.add("letraInicialDim", "a");
// (1)
```

```
    DrillSideways ds =
            new DrillSideways(searcher, config, taxoReader);
    // (2)
    DrillSidewaysResult drillResult = ds.search(query, 10);
    int qtdCategorias = 3;
    System.out.println(
            drillResult.facets.getAllDims(qtdCategorias));
```

A próxima listagem mostra o resultado com o *drill sideways* aplicado na dimensão `letraInicialDim=a`. O número de itens em `letraInicialDim` é 15708, então não foi aplicado nenhum filtro aqui, por isso ainda aparecem todos os valores desta dimensão. Mas nas demais dimensões o número é 591, ou seja, o filtro foi aplicado e há 591 produtos com inicial "a".

```
== DrillSideways ==
[dim=letraInicialDim path=[] value=15708 childCount=33
  c (4979)
  p (1794)
  m (1612)
, dim=categoriaDim path=[] value=591 childCount=43
  Cabos e Adaptadores (116)
  Antenas (97)
  Adaptadores de Rede (53)
, dim=produtoNomeDim path=[] value=591 childCount=44
  adaptador (222)
  antena (108)
  apple (72)
]
```

Fica mais simples quando comparamos esse *drill sideways* com o *drill down* a seguir. Perceba que as 3 dimensões (`categoriaDim`, `letraInicialDim` e `produtoNomeDim`) têm a mesma quantidade de itens: 591. Além disso, a `letraInicialDim` tem apenas o valor "a", assim, em uma tela não seria possível para o usuário selecionar um valor diferente.

```
== DrillDown==
[dim=categoriaDim path=[] value=591 childCount=43
  Cabos e Adaptadores (116)
```

```
  Antenas (97)
  Adaptadores de Rede (53)
, dim=letraInicialDim path=[] value=591 childCount=1
  a (591)
, dim=produtoNomeDim path=[] value=591 childCount=44
  adaptador (222)
  antena (108)
  apple (72)
]
```

Resumo

Neste capítulo foram apresentados recursos avançados usados como complementos nos sistemas de busca. São recursos que fazem parte do cotidiano, funções que podem nem ser consideradas novas para alguns tipos de usuário. Podemos dizer que o uso de sinônimos, *highlight* e o *More Like This*, por exemplo, são parte da vida de um usuário do Google ou Bing.

Da mesma forma, o corretor ortográfico e a sugestão de resultados permitem que nossa aplicação tenha uma interface muito próxima dos sites mais famosos de busca e de *e-commerce*, não limitada apenas à busca por palavra-chave.

Os recursos para análise de texto, como o cálculo da frequência dos termos e a indexação de vetores funcionam apenas no *back-end* da aplicação, quer dizer que não têm interface gráfica. Podem ser usados para explorar o conteúdo textual como um primeiro passo para a implantação de algoritmos de inteligência artificial.

Fechando o capítulo, uma funcionalidade essencial nos buscadores modernos: *facets*. Os *facets* são uma forma de categorização do resultado da consulta. Eles facilitam a navegação no site, tornando a experiência mais agradável para o cliente.

Capítulo 10
Extraindo dados da internet

Neste capítulo serão mostradas as técnicas para aquisição e extração de dados em sites. A primeira é o web *crawling*, onde é feito o rastreamento e a cópia dos dados das páginas, e a segunda é o web *scraping*, onde temos a extração de partes específicas das páginas. Este tema não é diretamente ligado ao Lucene, mas está em uma área comum, que é o processamento de texto.

Essas técnicas para extração de dados da internet são necessárias porque o texto de um site está escrito em linguagem natural, ou seja, de uma forma que o ser humano consegue ler, mas que um computador tem dificuldade para entender. Para que o computador entenda, precisamos fazer um pré-processamento através do web *crawling/scraping*.

Dizemos que a internet é a web de documentos, em que as páginas são, geralmente, documentos, isto é, a internet é composta por páginas HTML. E para isso temos linguagens (HTML, HTML5, XHTML, JavaScript), *frameworks* (jQuery, Angular, React, Ember), folhas de estilos (CSS), enfim, uma infinidade de recursos para criar sites agradáveis. Uma simples página web pode ter milhares de linhas, uma verdadeira confusão de código misturado com o texto.

Este capítulo é um pouco mais avançado e o código de exemplo usa lógica e recursos mais complexos do que no restante do livro. Vamos aprender como extrair dados de texto, identificar padrões e navegar entre elementos da página HTML, o que torna necessário um conhecimento básico de DOM e CSS. Uma leitura rápida em *Introdução ao DOM* (https://developer.mozilla.org/pt-PT/docs/Gecko_DOM_Reference/Introduction/) e em *CSS básico* (https://developer.mozilla.org/pt-BR/docs/Aprender/Getting_started_with_the_web/CSS_basico/) é suficiente.

Devido a essa complexidade, o processamento é feito em diversas fases, umas mais simples e outras, nem tanto. Vamos usar uma sequência de passos ordenados para facilitar o entendimento, uma área chamada de *Information Extraction* (IE), ou extração de informação. Mas tenha em mente que esta é uma das formas de realizar essas atividades, bem como há várias outras ferramentas além das listadas. O *Stanford NLP Group* (https://nlp.stanford.edu/) tem pesquisas e projetos avançados, disponibilizando ferramentas e documentação sobre IE.

Vale citar que existe uma iniciativa chamada de *Web Semântica* (http://www.w3c.br/Padroes/WebSemantica/), ou web dos dados, na qual está sendo implementado um padrão comum de formato de dados e protocolos de transferência. Os mais conhecidos são o *Resource Description Framework* (RDF) e a *Web Ontology Language* (OWL). Com a Web Semântica será mais fácil compartilhar e reutilizar dados entre aplicações e empresas. De uma forma mais simples, pode-se dizer que a internet será organizada para formar um grande banco de dados. Será uma nova maneira de representar e compartilhar informações.

10.1 Web crawling

A primeira parte do nosso projeto é a aquisição dos dados e será feita por meio de um web *crawler*, também chamado de web *spider*, robô de internet ou apenas de *bot*, em alusão à palavra inglesa *robot*; que é um tipo de software que rastreia a internet através dos links de cada página e extrai o seu conteúdo textual para que o buscador faça a indexação. Naturalmente, isso pode levar a um volume imenso de processamento, como no caso do Google ou do Bing, que rastreiam todas as páginas da internet e ainda analisam o conteúdo. É o que faremos neste capítulo, claro, em uma escala bem menor.

O *crawling* geralmente está ligado à internet, mas pode ser feito em intranet e até mesmo na sua máquina local. Uma vez que o conteúdo é recuperado através do *crawler*, o próximo passo é indexá-lo, ou, como veremos mais tarde no capítulo, podemos também utilizar algumas técnicas de processamento de linguagem (PLN) para fazer análises nos textos extraídos.

As análises incluem a **classificação** de páginas web dentro de categorias como esporte, política e economia. Então, quando aparece um texto sobre determinado tema de interesse do usuário, o sistema realiza uma ação, avisando o usuário ou apenas criando um painel de notícias, assim como o *Flipboard* (https://flipboard.com/).

Há também a **análise de sentimentos**, na qual o conteúdo, geralmente comentários de redes sociais, é classificado em positivo, neutro ou negativo, claro, de acordo com o sentimento de quem o escreveu. É uma boa forma de avaliar as opiniões sobre produtos ou sobre políticas. Se há muitas notas negativas, é um indicativo de que algo deve ser alterado. É o tipo de ferramenta usada por políticos nas últimas eleições.

Outra possibilidade é a **comparação** de preços entre sites. Aqui, um agente rastreia vários sites de *e-commerce*, extrai os produtos e seus respectivos preços para criar um buscador especializado em produtos. Atualmente há APIs de integração que permitem a troca desse tipo de informação entre empresas, ainda assim, você pode construir o seu próprio comparador de preços.

Uma aplicação um pouco mais avançada faz o **processamento de dados jurídicos**. São as chamadas *lawtechs*, empresas de tecnologia e direito que usam Inteligência Artificial (IA) para tentar auxiliar e agilizar o trâmite judicial dos processos. Demanda um pouco mais de trabalho porque envolve conhecimento sobre direito para o treinamento e personalização da solução. No Brasil existe a *AB2L* (https://www.ab2l.org.br/), uma associação de *lawtechs* e

legaltechs, na qual você pode encontrar o que essas empresas estão fazendo.

Em todos os casos, estamos falando de extração de conteúdo textual, pré-processamento e, então, indexação. É a mesma estratégia usada nos grandes buscadores, mas aqui será vista em pequena escala. Vai exigir um pouco de conhecimento de linguagens para web, principalmente HTML e CSS, bem como expressões regulares e uma boa dose de lógica de programação para conseguir extrair exatamente a informação relevante de cada página. Para entender mais, leia o artigo *The Anatomy of a Large-Scale Hypertextual Web Search Engine* (http://infolab.stanford.edu/~backrub/google.html), escrito pelos criadores do Google, onde eles mostram a arquitetura inicial do buscador. Veja em especial a seção **Crawling the Web**, que fala das dificuldades encontradas durante a implementação das ferramentas.

Apache Nutch

O ecossistema do Lucene tem uma ferramenta para web *crawling*, chamada de *Apache Nutch* (http://nutch.apache.org/). O Nutch é composto por plugins, o que permite um alto nível de modularização. Há, por exemplo, módulos nativos para indexação com o Apache Solr ou Elasticsearch, produtos não cobertos neste livro. Não há um plugin nativo para Lucene (incrível, mas é verdade), por isso vamos criar uma aplicação que leia o *dump* do Nutch e crie um índice diretamente no Lucene.

A instalação é bastante simples, você precisa descompactar o conteúdo do arquivo binário (`apache-nutch-1.X-bin.zip`) em um diretório de sua preferência. Para referência no capítulo, vamos chamar este diretório de instalação de `NUTCH_HOME` . A versão utilizada nestes exemplos é a 1.13, que conta com um servidor REST e interface web.

Internamente, o Nutch usa o Apache Hadoop, um framework para processamento em lote de grandes volumes de dados. É provavelmente a ferramenta mais conhecida para implementar soluções *big data*. O Hadoop permite a execução de programas em *clusters* com diversos servidores em paralelo, o que acelera o tempo de processamento. Um dos casos de uso do Hadoop é exatamente a indexação de páginas web, o que ele faz com eficiência.

Um web *crawler* funciona em estágios. O primeiro passo é buscar a lista de URLs que serão rastreadas. Essa lista é passada ao *fetcher*, que baixa o conteúdo das páginas, se possível, em paralelo através de *threads*. O passo final é gravar o conteúdo das páginas e os seus metadados (data de rastreamento, servidor web, tipo e tamanho do conteúdo etc.). Veja a ilustração:

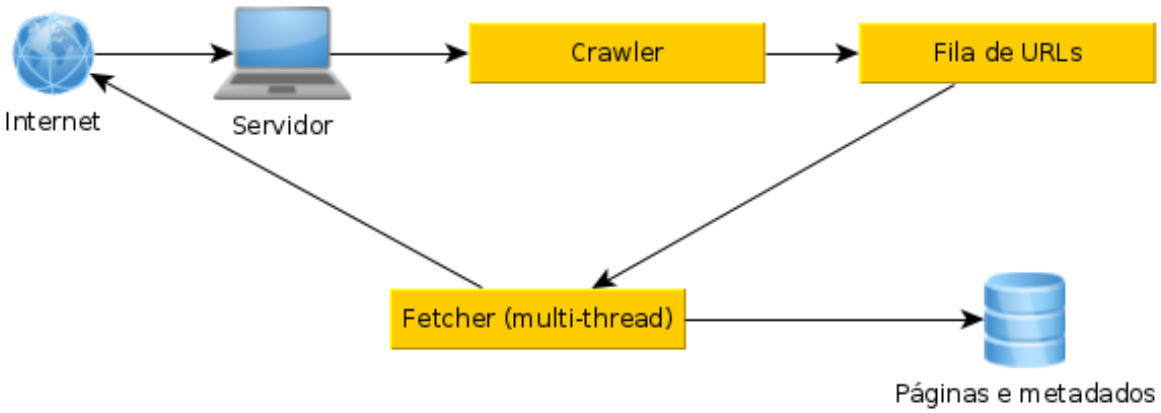


Figura 10.1: Arquitetura de um web crawler

É através dos web *crawlers* que os buscadores adquirem o conteúdo da internet. Cada buscador tem um ou vários *crawlers*, que são chamados de agentes, e tem um identificador próprio. Com isso, temos um agente do Google, um do Bing, um do DuckDuckGo etc. Os principais identificadores de agentes são Googlebot (Google), Bingbot (Bing), Slurp (Yahoo!), DuckDuckBot (DuckDuckGo) e Baiduspider (Baidu). Esse identificador é importante porque um site pode escolher quais agentes têm permissão de acessar seu conteúdo. Cada site pode definir suas

permissões para *crawling*, que estão definidas em http://www.robotstxt.org/.

Você pode conferir as permissões de rastreio em qualquer site através da URL `www.nome-do-site.com.br/robots.txt`, por exemplo, no caso do meu site pessoal seria `http://blog.marcoreis.net/robots.txt`. As principais configurações estão listadas logo na sequência. No bloco de código a seguir temos a configuração mais restritiva, na qual simplesmente não é permitido minerar os dados da página através de *bots*. Veja que o `User-agent` é definido como `*`, então todos os agentes estão desabilitados (`Disallow: /`) para acessar qualquer conteúdo desta URL:

```
User-agent: *
Disallow: /
```

O arquivo de `robots.txt` a seguir desabilita apenas o que estiver atrás de `/admin` e `/user`, mas permite todo o resto com o `Allow: *`. O `Craw-Delay` indica que o *crawler* deve esperar 5 segundos entre chamadas ao site:

```
User-agent: *
Disallow: /admin/
Disallow: /user/
Allow: *
Crawl-Delay: 5
```

Por fim, você pode querer permitir apenas um agente específico, como o Googlebot, desta forma:

```
User-agent: Googlebot
Allow: *
```

Configurando o crawler

A maneira mais simples de executar um *job* Nutch é através da linha de comando no *prompt*. Inicie por definir o diretório do Java através da variável de ambiente `JAVA_HOME`. Cada sistema operacional faz isso de uma forma diferente. No Windows, vá ao `Painel de Controle -`

`> Configurações avançadas de sistemas -> Variáveis de ambiente -> Variáveis de sistema .`

No Linux, apenas digite o comando `export JAVA_HOME={seu-diretorio-de-instalacao-do-java}` . Se for executar várias vezes o comando, talvez seja o caso de configurar essa variável automaticamente no *login* do usuário.

Precisamos mudar alguns parâmetros no agente que fará o rastreamento. Essas configurações estão no arquivo `NUTCH_HOME/conf/nutch-site.xml` :

- `http.agent.name` : o nome do seu agente, algo como `NutchBookLucene` ;
- `http.agent.email` : use o formato `nome dot sobrenome at seu-dominio dot com` , por exemplo, `marco.reis@meusite.com` ;
- `http.agent.description` : uma breve descrição do propósito do seu agente;
- `http.content.limit` : limite de bytes do conteúdo para download. Por padrão é 65 KB. Use -1 para baixar páginas de qualquer tamanho;
- `fetcher.server.delay` : tempo em segundos entre chamadas do *crawler*;
- `fetcher.threads.per.queue` : número de *threads* simultâneas baixando conteúdo (leia a seção sobre *10.4 Considerações legais e éticas*).

O ponto mais importante de um *crawler* é a lista de URLs que se pretende rastrear. Antes de escolher os sites, leia a seção sobre *10.4*, porque algumas páginas não concedem permissão de rastreamento através de *crawlers*. No nosso caso, vamos rastrear o meu site pessoal, no qual estão páginas projetadas para esta atividade. Depois de entender como funciona a técnica, você pode aplicar a qualquer outro site que permita.

Nosso exemplo prático vai rastrear o site `http://blog.marcoreis.net` . Para tanto, crie um diretório chamado `NUTCH_HOME/seed` e dentro deste

crie o arquivo texto `urls` , que vai conter as URLs que pretendemos rastrear. Neste arquivo, apenas escreva `http://blog.marcoreis.net` . Significa que o Nutch vai abrir essa página e procurar pelos links internos, como veremos na próxima seção.

Mas não vamos começar o rastreamento ainda. Note que uma página web pode ter (e geralmente tem) muitos links externos, que são irrelevantes para nosso objetivo de indexar apenas o conteúdo do nosso site. Assim, precisamos filtrar os links que apontam para sites externos. Isso é feito no arquivo `NUTCH_HOME/conf/regex-urlfilter.txt` , que tem uma lista de filtros predefinidos. Neste arquivo, vamos adicionar mais alguns filtros e depois permitir apenas as URLs que contenham `marcoreis` . Vamos tirar do *crawler* as páginas de comentários, *feed*, categoria etc. Depois de rastrear alguns sites você vai ver que é importante tem um bom filtro para retirar páginas indesejadas.

Veja como fica a parte final do arquivo de configuração:

```
# accept anything else
# +.
# filtra páginas indesejadas
-/comments/
-/feed/
-/[0-9]{4}/
-/category/
-/page/
-/tag/
+marcoreis
```

DICA: para testar se uma URL será rastreada, use o comando `bin/nutch plugin urlfilter-regex org.apache.nutch.urlfilter.regex.RegexURLFilter`. Depois de executá-lo, digite a URL que pretende avaliar e pressione `enter`. Como resultado, o programa mostra um sinal de `+` ou `-`, indicando se a URL será ou não rastreada. Se estiver no Linux, pode executar o comando `cat seed/urls | bin/nutch plugin urlfilter-regex org.apache.nutch.urlfilter.regex.RegexURLFilter`, que ele valida todas as URLs cadastradas no Nutch.

Comandos para crawling

O *crawler* está preparado para começar a trabalhar. Faremos um rastreamento sem profundidade apenas para experimentar e confirmar que tudo está funcionando. É importante notar que todos os comandos são executados no *prompt* de comando a partir do diretório `NUTCH_HOME`. No Windows ou Linux use o comando `cd NUTCH_HOME` para entrar no diretório correto. A partir dele execute os comandos listados a seguir.

O primeiro passo é o rastreamento, que será gravado no diretório `crawl`. Após sua conclusão serão criados 3 subdiretórios: (1) `crawldb`, banco de dados que registra o status de rastreamento de cada página; (2) `linkdb`, banco de dados que registra os *links* de origem; (3) `segments`, cada iteração (você vai entender daqui a pouco) produz um diretório de segmento com o conteúdo das páginas. Agora, vamos para os comandos:

(1) Para iniciar o rastreamento use o comando: `bin/crawl seed crawl 3`. Este comando tem 4 parâmetros:

- **bin/crawl**: este é o programa que inicia processo de *crawling*, ou rastreamento;
- **seed**: diretório no qual estão as listas de URLs;
- **crawl**: diretório de saída que será criado com o resultado do processo;

- **Quantidade de iterações (3)**: indica a quantidade de iterações de rastreamento.

A primeira iteração rastreia os links da página inicial, a segunda rastreia os links internos da inicial e assim por diante. Se você usar 10 iterações, a quantidade de páginas visitadas será exponencialmente maior, então use com cautela. Os dados do rastreamento são gravados no disco com o formato do Hadoop, um tipo de arquivo binário de alta performance. Entretanto, não é possível ler seu conteúdo porque ele não é textual. Mais tarde nesta seção veremos como transformar os dados em texto.

(2) Para mostrar as estatísticas do rastreamento use o comando: `bin/nutch readdb crawl/crawldb/ -stats` . Aqui podemos analisar os números do rastreamento realizado pelo *crawler*. Os parâmetros usados são:

- **bin/nutch**: programa que contém vários utilitários, como este que mostra estatísticas;
- **readdb**: parâmetro para ler o registro de páginas rastreadas;
- **crawl/crawldb**: diretório no qual guardaremos as páginas rastreadas;
- **-stats**: parâmetro para mostrar as estatísticas do rastreio.

Estão disponíveis dados sobre o número de URLs e o status. A lista a seguir mostra as opções de status e a quantidade de páginas em casa situação:

```
status 1 (db_unfetched): XX
status 2 (db_fetched): XX
status 3 (db_gone): XX
status 4 (db_redir_temp): XX
status 5 (db_redir_perm): XX
status 7 (db_duplicate): XX
```

(3) Para listar os segmentos gerados use o comando: `bin/nutch readseg -list -dir crawl/segments` . A lista dos segmentos mostra os 3

diretórios resultantes da execução do rastreamento. Os parâmetros são:

- **bin/nutch**: utilitário;
- **readseg**: parâmetro para ler o diretório de segmentos;
- **-list**: parâmetro que indica que o resultado será mostrado na tela;
- **-dir craw/segments**: parâmetro que indica o diretório em que estão os segmentos.

O resultado é algo semelhante a esta listagem:

NAME	GEN	START	END	FET
20171031144252	65	2017-10-31T14:42:56	2017-10-31T14:44:58	65
20171031144051	1	2017-10-31T14:40:56	2017-10-31T14:40:56	1
20171031144110	45	2017-10-31T14:41:15	2017-10-31T14:42:36	45

(4) Para gerar o *dump* das URLs: `bin/nutch readdb crawl/crawldb -dump crawl/dump-crawldb` . O *dump* das páginas mostra mais detalhes sobre o processo de rastreamento de cada página. Os parâmetros são:

1. **bin/nutch**: programa utilitário;
2. **readdb**: parâmetro para ler o registro de páginas;
3. **crawl/crawldb**: diretório das páginas;
4. **-dump**: indica que será criado um *dump* dos dados no diretório `crawl/dump-crawld` .

Como resultado da execução, será gravado ao menos um arquivo chamado `part-00000` com os detalhes da execução. O próximo comando faz o *merge* dos segmentos. Como foram gerados 3 diretórios de segmentos, vamos juntá-los em um único diretório com o próximo comando.

(5) Para juntar todos os diretórios de segmentos: `bin/nutch mergesegs crawl/merged crawl/segments/*` . O diretório `crawl/merged` foi criado pela opção `mergeset` , que contém a união de todos os segmentos, e será usado para gerar o *dump* do conteúdo textual das páginas rastreadas. Este conteúdo servirá de base para o web *scraping*.

(6) Para gerar o *dump* do conteúdo baixado: `bin/nutch readseg -dump crawl/merged/* crawl/dump-conteudo` . O parâmetro inédito é o `-dump` , que gravará os dados em formato textual no arquivo `dump` dentro do diretório `crawl/dump-conteudo` . A partir daqui, os comandos são opcionais e podem ser usados para exploração de conteúdo.

(7) Para gerar o *dump* das páginas: `bin/nutch dump -outputDir crawl/dump-pages -segment crawl/merged/` . Exporta o conteúdo rastreado em formato HTML. Cada página rastreada será gravada de volta em um arquivo HTML.

(8) Dump dos links: `bin/nutch readlinkdb crawl/linkdb/ -dump dump-linkdb` . Gera o *dump* dos links que geraram o rastreamento.

(9) Verificador de URLs: `bin/nutch parsechecker http://blog.marcoreis.net/` . Este é um ótimo utilitário que lê a página e analisa seu conteúdo, mostrando seus links e metadados.

(10) Verificador com *dump* do conteúdo: `bin/nutch parsechecker -dumpText http://blog.marcoreis.net` . O comando é uma variação do (9), mas agora mostra também o texto da página. Com isto, terminamos a seção de web *crawling*. A próxima seção mostra como analisar e recuperar informações dentro do HTML através da técnica de web *scraping*.

10.2 Web scraping

O web *scraping*, ou raspagem de dados da web, é uma técnica na qual uma ferramenta extrai partes relevantes de um texto

proveniente da internet. É uma técnica que não se aplica unicamente à web, nem precisa de um programa. Uma pessoa e um *browser* podem fazer web *scraping*, entretanto, nesta seção tratamos apenas de raspagem de dados em páginas da web através de um programa automatizado.

É uma técnica diferente do web *crawling*. Consideramos que no web *scraping* vamos extrair e analisar partes específicas de uma página, enquanto no web *crawling* nós rastreamos e baixamos o conteúdo completo de diversas páginas web, sem nenhum tipo de análise prévia. São recursos usados, geralmente, em conjunto, principalmente porque uma grande parte das informações está na internet. Mas nada impede que façamos *scraping* em PDFs e DOCs, por exemplo, à procura de nomes de pessoas e lugares. Igualmente, você pode fazer o web *crawling*, baixar as páginas e indexá-las sem nenhuma análise. Ainda assim, alguns autores podem se referir às duas técnicas como se fossem apenas uma, ou até mesmo chamar tudo de *bots*. Eu prefiro separar os conceitos por uma questão de didática. Concluímos com isso que web *crawling*, web *scraping*, indexação e PLN são recursos diferentes e independentes, mas que têm sido usados em conjunto.

Durante uma navegação ocasional pela internet, um usuário tem acesso limitado à informação, porque só pode ver uma página de cada vez, perdendo várias oportunidades. Com um web *crawler* um sistema tem acesso a milhares de páginas, e depois do web *scraping* podemos resumir ou agrupar essa informação. Não à toa, temos várias ferramentas que usam essas técnicas para criar novas oportunidades, especialmente se associarmos os dados com a data de sua coleta e sua localização, criando uma linha temporal e uma posição no mapa. Você já deve ter visto isso: um site que mostra a variação de preço de um produto entre lojas ao longo do tempo.

Claro, seria muito mais fácil baixar os dados por meio de APIs, mas nem sempre existe uma disponível que seja adequada à sua necessidade. Nosso país tem uma boa iniciativa com o *Portal Brasileiro de Dados Abertos* (http://dados.gov.br/). Ainda assim,

apenas uma pequena porção da web está disponível via APIs, o que confere uma longa vida às ferramentas de web *scraping*.

O web *scraping* é possível porque as páginas são baseadas em HTML, uma linguagem de marcação, ou seja, existem marcas que definem os seus elementos. Há uma marca (ou *tag*) para definir o título, o corpo do texto, os parágrafos etc. Essa estrutura garante que as páginas sejam legíveis para um usuário, mas trazem um certo nível de complexidade para nosso web *scraper*.

Como visto no começo do capítulo, as ferramentas também podem usar DOM (Document Object Model) ou CSS (Cascading Style Sheet). O DOM é uma árvore de objetos construída a partir dos elementos de uma página HTML, que pode ser conferido na imagem abaixo. O CSS é uma linguagem que descreve os estilos usados na página. Vale lembrar que temos as referências *Introdução ao DOM* (https://developer.mozilla.org/pt-PT/docs/Gecko_DOM_Reference/Introduction) e *CSS básico* (https://developer.mozilla.org/pt-BR/docs/Aprender/Getting_started_with_the_web/CSS_basico), sendo estas duas o suficiente para o nosso capítulo.

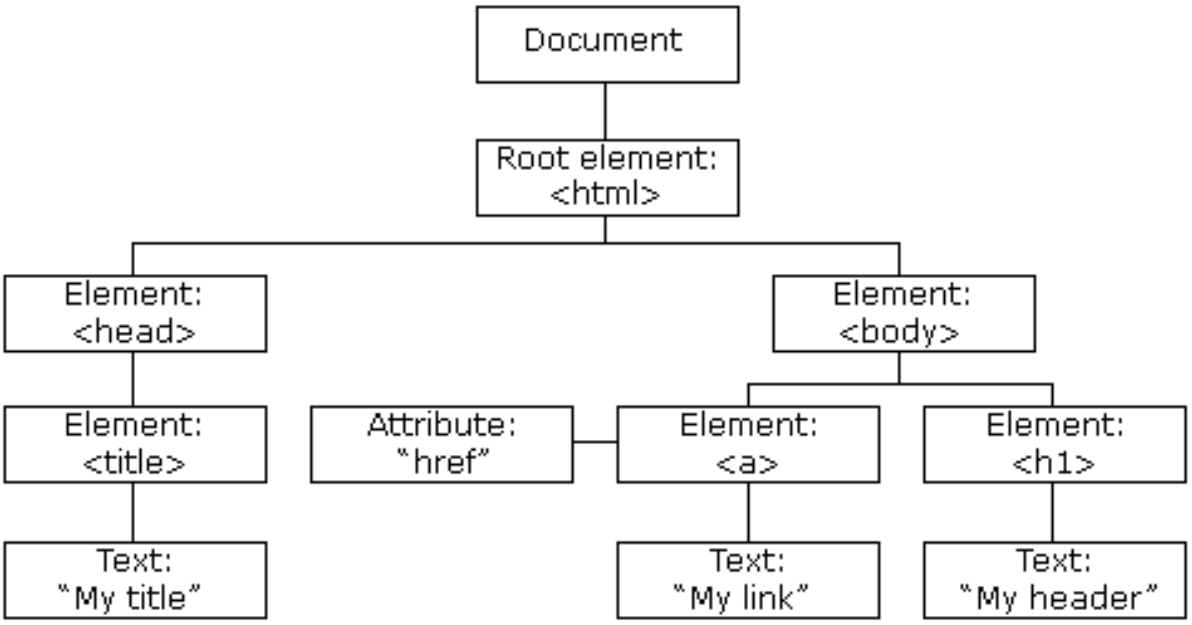


Figura 10.2: Árvore de objetos DOM. Crédito da imagem: W3schools.

Essas opções não são as únicas. Também é possível usar os *feeds* dos sites, ou até mesmo Inteligência Artificial, por meio de técnicas de *Machine Learning*, que não serão vistas aqui no livro, mas há bastante material disponível gratuitamente, como pode ser conferido no capítulo de referências bibliográficas, em que está listado o livro da Ryan Mitchell sobre web *scraping*.

Inspetor de páginas

Baixamos várias páginas web e agora vamos escolher, dentro delas, quais os dados serão guardados e o que será descartado. Vamos analisar as páginas baixadas e identificar os elementos que são úteis para nosso programa e o que é desnecessário. O conteúdo principal da página será guardado, enquanto os textos laterais, rodapé e cabeçalhos serão descartados. A menos, claro, que para sua aplicação esses elementos tenham alguma relevância.

Como o exemplo usa páginas de um blog, a grande maioria do código HTML é descartado, enquanto apenas a área legível para o usuário será utilizada. O princípio é o mesmo para um site de

notícias ou de compras. O texto da notícia ou o produto são apenas uma pequena parte da página, enquanto a grande maioria do código é descartável. Para analisar as páginas, uma ótima ferramenta é o *Page Inspector do Firefox* (https://developer.mozilla.org/en-US/docs/Tools/Page_Inspector/UI_Tour), ou o *Chrome DevTools* (https://developers.google.com/web/tools/chrome-devtools/).

Uma página HTML bem formada deve conter uma estrutura parecida com isso:

```
<html>
    <head>
        <title>Sou um título</title>
        <!-- outros scripts -->
    </head>

    <body>
        <h1>E eu sou um header</h1>
        <div id="minha-div">Aqui jaz uma div</div>
        <!-- vários outros elementos -->
    </body>
</html>
```

Para abrir as ferramentas, pressione as teclas `control + shift + c` e o inspetor vai aparecer no *browser*. A imagem a seguir mostra o inspetor do Chrome, que será aberto:

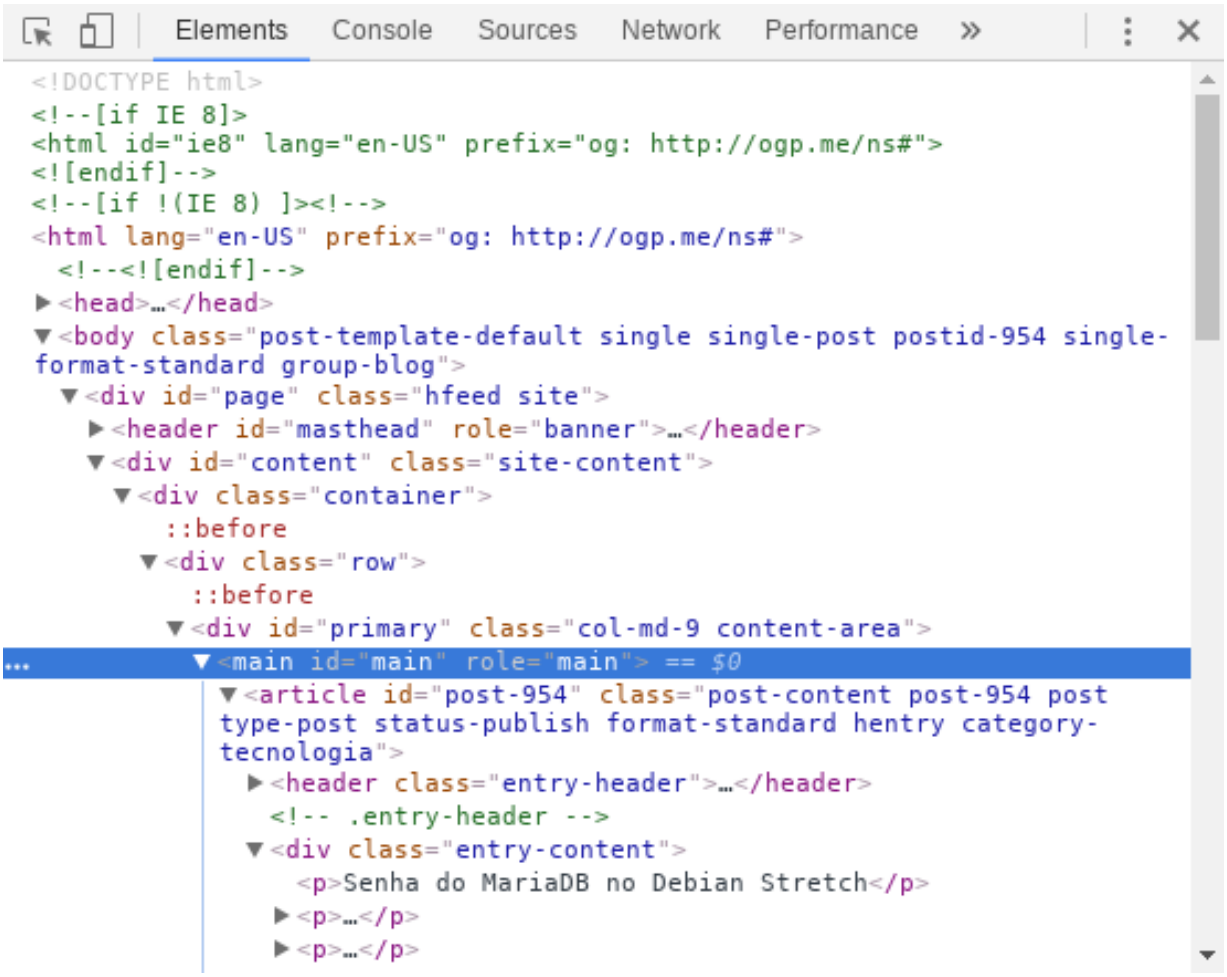


Figura 10.3: Chrome DevTools

Quando passamos o mouse sobre os elementos do inspetor ou da página, você vai perceber que essa parte específica da página fica azul. Por exemplo, considerando a última imagem, na qual está selecionado o elemento `<main id="main" role="main">`, você vai perceber que essa parte da página fica azul, como na figura a seguir.

Figura 10.4: Seleção do elemento na página

O inspetor de páginas do Firefox funciona da mesma forma e, como visto na imagem a seguir, tem a mesma aparência. Em termos de funcionalidade, são equivalentes.

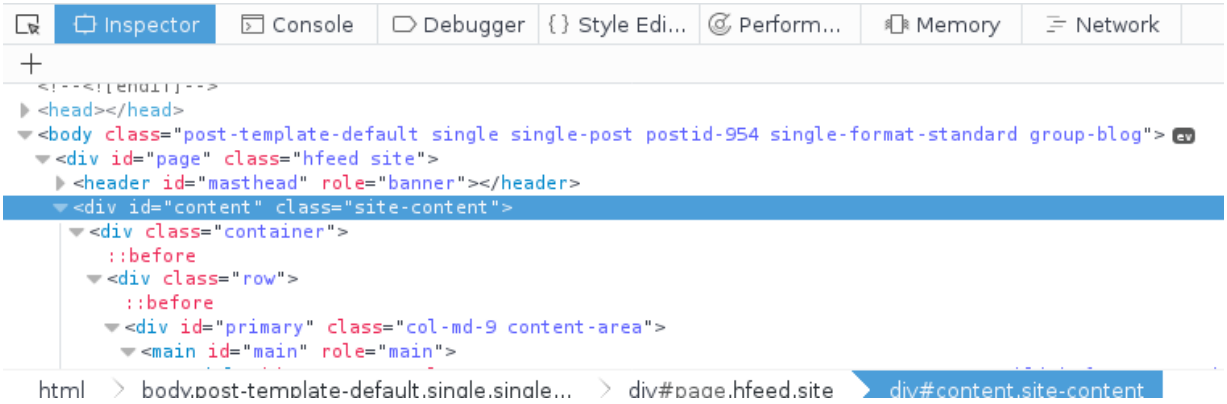


Figura 10.5: Firefox Page Inspector

Ao analisar as páginas de diversos sites, fica claro que há muitas diferenças, então, para cada site, você precisa de uma estratégia diferente. No nosso exemplo, o conteúdo principal está contido em `<main id="main" role="main">`, em outro site certamente será diferente. Isso sem contar com o fato de que os sites mudam, e toda a análise precisa ser refeita. Por exemplo, quando um e-commerce troca de plataforma, o código HTML vai mudar, mesmo que visualmente esteja igual. Neste caso, temos de atualizar o nosso web *scraper*.

Web scraping em texto

O conteúdo das páginas HTML que foi baixado pelo *crawler* foi convertido em um arquivo texto. Esse *dump* está gravado em `NUTCH_HOME/crawl/dump-conteudo/dump` e contém o código de todas as páginas, além de mais alguns metadados. Após uma breve análise, você vai perceber que há determinados padrões neste arquivo. Esta é a chave para se fazer a raspagem dos dados: descobrir os padrões e usar as ferramentas para extrair seus valores. O *dump*, por exemplo, é dividido em blocos com delimitadores e, se olhar com cuidado, vai perceber que há separadores bem definidos, que estão listados a seguir. Veja que eles terminam com `::` (dois pontos repetidos). Este é o tipo de referência que deve ser percebido.

- `Recno::` - número sequencial da URL que foi rastreada;
- `CrawlDatum::` - dados sobre o rastreamento;
- `ParseData::` - metadados da página, incluindo os links;
- `ParseText::` - texto extraído da página;
- `URL::` - a própria URL;
- `Content::` - este é o ponto central, que é o código HTML da página.

Dentro do *dump*, o separador de páginas é o atributo `Recno::`, então, após cada `Recno::`, temos o conteúdo de uma página diferente, bem como quantas vezes ela foi visitada. O próximo separador importante para nosso agente é o `Content::`. É importante notar que nem toda URL tem conteúdo HTML. A partir dele temos os dados do código HTML da página. Com o conteúdo podemos fazer todo tipo de processamento, como simplesmente indexar o texto; procurar por produtos e preços; ou aplicar algoritmos de IA.

Para aplicar as técnicas, neste capítulo foi criado o projeto `commons-web-crawler` no *GitHub* (https://github.com/masreis/commons-web-crawler/) com diversos utilitários. O primeiro é o `DumpNutchParser`, que transforma o *dump* das páginas em uma lista de objetos `DumpNutchVO`. A partir deste, podemos proceder com as demais análises da página. Os campos definidos são `url`, que guarda, claro, a URL da página; o `content`, que é o próprio conteúdo HTML de cada página;

`contentType` guarda o tipo de conteúdo; e o `metadata`, que são os metadados das páginas.

```
public class DumpNutchVO {
    private byte[] url;
    private byte[] content;
    private byte[] contentType;
    private byte[] metadata;
}
```

A classe `DumpNutchParser` extrai determinados blocos de texto do *dump* para montar a lista de `DumpNutchVO`. Ela recebe como parâmetro o caminho do arquivo do *dump*, extrai o texto, separa os blocos de texto com o `Recno::` e verifica se aquela URL tem conteúdo HTML (`Content::`). Caso positivo, ele cria um objeto com o conteúdo da página. Veja:

```
public class DumpNutchParser {
    private static final Logger logger =
            Logger.getLogger(DumpNutchParser.class);

    public List<DumpNutchVO> parse(File arquivoDump)
            throws IOException {
        List<DumpNutchVO> lista = new ArrayList<DumpNutchVO>();
        BufferedReader buf =
                new BufferedReader(new FileReader(arquivoDump));
        String linha = null;
        StringBuilder conteudo = new StringBuilder();
        buf.readLine();
        buf.readLine();
        while ((linha = buf.readLine()) != null) {
            if (linha.startsWith("Recno::")) {
                if (conteudo.toString().contains("Content::")) {
                    DumpNutchVO vo = criaVO(conteudo.toString());
                    if (vo != null) {
                        lista.add(vo);
                    }
                }
                conteudo.setLength(0);
            } else {
```

```
                conteudo.append(linha).append("\n");
            }
        }
        buf.close();
        return lista;
    }
}
```

A estrutura do *dump* é parecida com o bloco de texto adiante. O método `criaVO`, visto em seguida, recebe este bloco como parâmetro de entrada e vai usar expressões regulares para extrair o conteúdo.

```
{...}
Content::
Version: -1
url: http://blog.marcoreis.net/
base: http://blog.marcoreis.net/
contentType: text/html
metadata: Server=Apache/2.2.29 (Unix) mod_ssl/2.2.29 OpenSSL/1.0.1e-fips
mod_wsgi/4.4.22 Python/2.6.6 mod_bwlimited/1.4 Connection=close Date=Mon,
06 Nov 2017 19:41:52 GMT nutch.crawl.score=1.0
nutch.fetch.time=1509997357294 nutch.segment.name=20171106174232 Content-
Encoding=gzip Vary=Accept-Encoding Content-Length=10622 _fst_=33 Link=
<http://blog.marcoreis.net/wp-json/>; rel="https://api.w.org/" X-Powered-
By=PHP/5.4.40 Content-Type=text/html; charset=UTF-8
Content:
<! DOCTYPE html>
{...}
```

O primeiro passo é encontrar o começo do conteúdo, descartando o resto do bloco. Isso é simples, pois o conteúdo começa depois do termo `Content::`. O código é este:

```
    private DumpNutchVO criaVO(StringBuilder conteudo) {
        DumpNutchVO vo = new DumpNutchVO();
        // Encontra o início do bloco do conteúdo
        int inicioConteudo = conteudo.indexOf("Content::");
        String conteudoCompleto =
                conteudo.substring(inicioConteudo);
```

```
        {...}
    }
```

O primeiro valor analisado é o da URL, que é extraído através da expressão `(url:\\s)(.*)`. Este código separa a linha em dois blocos que estão entre os parênteses. O primeiro bloco é o termo `url:` e o segundo é o resto da linha. Com o comando `matcher.group(2)` recuperamos o conteúdo que está no segundo parênteses, ou seja, a URL. Veja o código:

```
DumpNutchVO vo = new DumpNutchVO();
// Extrai a URL por meio de expressão regular
Pattern patternUrl = Pattern.compile("(url:\\s)(.*)");
Matcher matcher = patternUrl.matcher(conteudoCompleto);
String url = null;
if (matcher.find()) {
    url = matcher.group(2);
}
vo.setUrl(url.getBytes());
```

O conteúdo HTML é um pouco diferente, porque não fica em uma única linha. Para extraí-lo vamos marcar a posição de início e de fim da página HTML. Uma opção está mostrada no código a seguir. Sabemos que o HTML começa logo depois de `Content:` e termina em `</html>`. Adicionei uma condição para retornar nulo se a página não estiver devidamente finalizada.

```
// Encontra o início e o fim do código HTML
String termoContent = "Content:\n";
int inicioHtml = conteudoCompleto.indexOf(termoContent);
inicioHtml += termoContent.length();
String termoFimHtml = "</html>";
int fimHtml = conteudoCompleto.indexOf(termoFimHtml);
// Se o código da página não está correto
if (fimHtml < 0) {
    return null;
}
String conteudoHtml =
        conteudoCompleto.substring(inicioHtml, fimHtml);
vo.setContent(conteudoHtml.getBytes());
```

Para testar o *parser* temos a classe `DumpNutchParserTest`, que está a seguir. Altere o caminho para o arquivo do *dump* de forma a apontar ao diretório correto.

```
public class DumpNutchParserTest {
    public void testParser() throws IOException {
        DumpNutchParser parser = new DumpNutchParser();
        File arquivoDump = new File("crawl/dump-conteudo/dump");
        List<DumpNutchVO> lista = parser.parse(arquivoDump);
        for (DumpNutchVO vo : lista) {
            System.out.println(vo.getUrl());
        }
    }
}
```

Com isso, temos as páginas devidamente organizadas em classes, a partir das quais vamos poder realizar diversos tipos de processamento. No tópico a seguir veremos como extrair dados de páginas web, rastreando determinados elementos.

Ferramenta para web scraping - jsoup

A ferramenta utilizada para fazer o web *scraping* é o *jsoup* (https://jsoup.org/), uma biblioteca escrita em Java para trabalhar com HTML, e sua API permite a extração e manipulação de dados das páginas, usando o DOM e o CSS. Podemos extrair os dados de uma URL diretamente da internet, através de um arquivo gravado localmente ou até mesmo por meio de uma String. Com o jsoup podemos navegar entre os elementos da árvore DOM da página ou pelos seletores CSS.

Ele permite até mesmo alterar o valor dos atributos do HTML, mas isso não é necessário para nosso trabalho. Aqui vamos procurar por determinados elementos dentro da página e recuperar seu valor. No caso de um blog, vamos procurar pelo título e pelo texto principal; em um e-commerce vamos procurar pelo produto e seu preço; e em um portal de notícias procuramos a notícia principal.

Para usar o jsoup, precisamos adicionar a dependência do Maven, que é:

```
<dependency>
    <groupId>org.jsoup</groupId>
    <artifactId>jsoup</artifactId>
    <version>1.10.3</version>
</dependency>
```

Há ferramentas similares em Java e em outras linguagens, principalmente em Python, que é bastante usada para este tipo de processamento. Estes são recursos *open-source*, gratuitos e que demandam conhecimento técnico. Entretanto, há opções pagas para web *scraping*, como o *Import.IO* (https://www.import.io/) ou o *WebHose* (https://webhose.io/), que são produtos com interface amigável, acessados através do próprio *browser* e não precisam do esforço da programação.

Continuando com o jsoup, o primeiro exemplo vai mostrar como extrair os dados do cabeçalho HTML de uma página, que contém *tags* como `meta`, `link`, `script`, `style`, `comments` e `title`. Para resolver essa questão, foi criada a classe `ExtratorConteudo`, que tem como atributo um objeto do tipo `org.jsoup.nodes.Document` para representar uma página HTML. Os métodos que valem citação são o `Documento.head()` e `Document.body()`. O primeiro representa os elementos do cabeçalho e o segundo, o corpo da página.

Cada elemento da página é representado por um objeto do tipo `org.jsoup.nodes.Element`, ou seja, cada *tag* é representada por um `Element`. Dessa forma, os *links*, imagens, *divs*, parágrafos, qualquer *tag* dentro da página terá um objeto `Element` correspondente, o que permite a recuperação de elementos por meio do seu identificador, classe CSS, nome de atributo, valor etc. Uma vez que você identifique o elemento, o jsoup fornece meios para recuperá-lo.

O primeiro passo para usar o jsoup é carregar a página HTML no objeto `Document`, o que pode ser feito com os seguintes métodos:

- `public static Document Jsoup.parse(URL url, int timeoutMillis)` : carrega o documento a partir de uma URL;
- `public static Document Jsoup.parse(File in, String charsetName)` : carrega a partir de um arquivo local;
- `public static Document Jsoup.parse(String html)` : carrega a partir do código HTML, que será a nossa escolha.

A classe `ExtratorConteudo` pode ser conferida logo a seguir. Em `carregaConteudoHtml` o jsoup preenche o atributo `documento` com o conteúdo da página. O `extraiConteudoHead` recupera o *head* da página, depois recupera os elementos internos com o `head.children()`. Na sequência, vamos apenas imprimir o nome da *tag* e seu conteúdo. Há elementos com conteúdo textual (`elemento.text()`); com código-fonte, como no caso de *scripts* e estilos (`elemento.data()`); e os demais elementos HTML que são *self-closing* ou *void-elements*, isto é, cujo conteúdo fica junto à própria *tag*.

```
public class ExtratorConteudo {
    private static final Logger logger =
            Logger.getLogger(ExtratorConteudo.class);
    private Document documento;

    public void carregaConteudoHtml(String conteudoHtml) {
        documento = Jsoup.parse(conteudoHtml);
    }

    public void carregaConteudoHtml(URL url)
            throws IOException {
        int timeout_cinco_segundos = 5000;
        documento = Jsoup.parse(url, timeout_cinco_segundos);
    }

    public void extraiConteudoHead() {
        Element head = documento.head();
        Elements elementos = head.children();
        for (Element elemento : elementos) {
            logger.info(elemento.tagName());
            if (elemento.hasText()) {
```

```
                logger.info("Text: " + elemento.text());
            } else if (elemento.data().length() > 0) {
                logger.info("Data: " + elemento.data());
            } else {
                logger.info(elemento);
            }
            logger.info("================");
        }
    }
}
```

Para testar esse componente usamos a `ExtratorConteudoTest`. Durante a inicialização vamos preencher o `DumpNutchParser` com o caminho do nosso *dump*, como já foi feito nos exemplos anteriores. Com isso, passamos diretamente para a utilização do novo extrator:

```
    public void testExtraiConteudoHead() {
        for (DumpNutchVO vo : listaVos) {
            ExtratorConteudo extrator = new ExtratorConteudo();
            extrator.carregaConteudoHtml(vo.getContent());
            extrator.extraiConteudoHead();
        }
    }
```

Da mesma forma, podemos usar o extrator para qualquer outro site. Neste bloco de código vemos como extrair o cabeçalho da Wikipedia.

```
    public void testParseFromUrl()
            throws MalformedURLException, IOException {
        ExtratorConteudo extrator = new ExtratorConteudo();
        String spec = "http://pt.wikipedia.org";
        extrator.carregaConteudoHtml(new URL(spec));
        extrator.extraiConteudoHead();
    }
```

Para extrair o conteúdo do corpo da página usamos o método `Document.body()`. A partir dele, temos todos os elementos do corpo montados em formato de árvore. O *body* é o tronco principal dividido em ramos que, por sua vez, se dividem em outros ramos, até chegar

aos elementos da ponta, geralmente, os componentes visuais de uma página. Veja que não estamos falando de arquivos binários como imagens ou PDFs, mas é possível extrair o conteúdo deles e adicioná-los ao *scraper*.

As páginas rastreadas nestes exemplos são baseadas em WordPress, uma plataforma bastante usada (a mais usada) para criação de sites. Em *blogs*, o conteúdo principal de um *post* fica, geralmente, em uma *div* com classe CSS `entry-content`, `entry`, `main-content` etc. É só usar o inspetor de páginas e verificar no seu caso onde é que fica. Em *blogs* é mais ou menos padronizado, mas em sites de notícias e e-commerce é bem diferente. Claro que você pode simplesmente pegar o *body* da página inteiro sem filtrar, contudo, dessa forma a precisão das análises é bem menor. O ideal é que se busque exatamente a informação necessária, neste caso, apenas o conteúdo central, e ignore todo o resto.

É o que mostra o próximo bloco, da classe `ExtratorConteudo`, que recupera apenas o texto dentro da classe `entry-content`. O método `Element.text()` retorna apenas o texto puro, sem nenhuma formatação ou *tags* HTML. O que não for texto é ignorado.

Neste caso, há apenas uma entrada com essa classe, por isso é seguro retornar seu texto. Mas pode acontecer de vários elementos terem a mesma classe. Então precisamos encontrar alguma forma de identificar unicamente o elemento.

```
public String extraiConteudoPrincipalBlog() {
    Element body = documento.body();
    Elements entryContent =
            body.getElementsByClass("entry-content");
    if (entryContent != null && entryContent.hasText()) {
        return entryContent.text();
    } else {
        return null;
    }
}
```

O teste (`ExtratorConteudoTest`), como pode ser visto a seguir, extrai o conteúdo principal de cada página do *dump*. Vale lembrar que nem toda URL tem texto visível e que, para visualização, estamos mostrando apenas os primeiros caracteres.

```
    public void testExtraiConteudoPrincipalBlog() {
        for (DumpNutchVO vo : listaVos) {
            ExtratorConteudo extrator = new ExtratorConteudo();
            extrator.carregaConteudoHtml(vo.getContent());
            String conteudo =
                    extrator.extraiConteudoPrincipalBlog();
            logger.info(vo.getUrl());
            if (conteudo != null) {
                logger.info(conteudo.substring(0, 120));
            } else {
                logger.warn(
                        "Esta URL não tem conteúdo principal");
            }
        }
    }
```

Extraindo a categoria das páginas

Vamos adicionar mais uma camada de refinamento ao nosso projeto. Extraímos o cabeçalho, depois, o conteúdo principal. Agora, vamos extrair a categoria de cada página. Em blogs, é comum que cada página tenha ao menos uma categoria, classe ou rótulo. Nas nossas páginas, a categoria aparece através do elemento adiante, onde há um link, um `rel` (relacionamento) e um texto associado. Assim:

```
<a href="http://blog.marcoreis.net/category/big-data/" rel="category
tag">Big Data</a>
```

Um site pode ser classificado como negócio, jogos, recreação, referência etc. A lista completa está disponível no Alexa (https://www.alexa.com/topsites/category/). E isso interfere em todos os aspectos, até em seu acesso porque uma empresa pode ter políticas para bloquear sites adultos ou redes sociais.

Para recuperar todas as categorias da página temos de encontrar os elementos que contêm o **atributo** `rel` com **valor** `category tag` , que estão dentro de `<main id="main" role="main">` , que tem o **id** `main` . Em negrito estão os campos que serão recuperados pelo jsoup. Novamente, para navegar entre os elementos use o inspetor de páginas, você vai chegar ao mesmo lugar. Perceba que há uma hierarquia de elementos, formando uma árvore DOM, que pode ser lida como **html -> body -> main -> href -> categoria**.

O código a seguir mostra como implementar a funcionalidade de navegação seguindo esse modelo. Para cada página, vamos extrair as categorias e guardá-las em um `Set` , uma coleção que não permite repetição, já que queremos saber quais as categorias diferentes. Os métodos usados para navegar no DOM são o `getElementById(String id)` , `getElementsByTag(String tag)` , `getElementsByClass(String className)` e o `getElementsByAttributeValue` . Com isso, dá para fazer bastante coisa, mas é possível fazer ainda mais (https://jsoup.org/cookbook/).

```
public Set<String> extraiCategoriasBlog(
        List<DumpNutchVO> listaVos) {
    Set<String> lista = new HashSet<String>();
    for (DumpNutchVO vo : listaVos) {
        carregaConteudoHtml(vo.getContent());
        Element body = documento.body();
        Element main = body.getElementById("main");
        if (main == null) {
            continue;
        }
        Elements categorias =
                main.getElementsByAttributeValue("rel",
                        "category tag");
        for (Element e : categorias) {
            lista.add(e.text());
        }
    }
    return lista;
}
```

Para testar, apenas passe as páginas para o extrator:

```
    public void testExtraiCategoriasBlog() {
        ExtratorConteudo extrator = new ExtratorConteudo();
        Set<String> categorias =
                extrator.extraiCategoriasBlog(listaVos);
        for (String categoria : categorias) {
            logger.info(categoria);
        }
    }
```

Extraindo artigos da Wikipedia

A Wikipedia é uma ótima fonte de informação, mesmo que muitos dados não sejam totalmente confiáveis. Mas para nossa função de *scraping* é perfeita. É uma grande coleção de documentos categorizados, que pode ser usada até como fonte para os futuros exemplos de IA. Temos um roteiro para rastreamento definido, que inclui o próprio rastreamento, a junção dos segmentos e a geração do *dump*.

Para a Wikipedia, teremos os diretórios `seedwiki` e `crawlwiki`, evitando os diretórios já criados. Isso se dá porque o extrator para cada site é diferente, então estou usando diretórios separados para não confundir. A URL da Wikipedia em português é `https://pt.wikipedia.org` e deve ficar no arquivo `seedwiki/urls`. Altere o `regex-urlfilter.txt` para permitir URLs apenas da versão em português adicionando a expressão `+^https://pt.wikipedia.org`, ou seja, só considere as páginas que comecem com este padrão.

Fique à vontade para usar os comandos de estatística, mas aqui, resumidamente, precisamos apenas de:

- `bin/crawl seedwiki crawlwiki 3` : rastreamento;
- `bin/nutch mergesegs crawlwiki/merged crawlwiki/segments/*` : junção dos segmentos;
- `bin/nutch readseg -dump crawlwiki/merged/* crawlwiki/dump-conteudo` : geração do *dump*.

Com o inspetor de páginas podemos conferir que o conteúdo principal das páginas na Wikipedia fica dentro da *div* `content` :

```
<div id="content" class="mw-body" role="main">
```

Isso é suficiente para extrair o conteúdo principal de qualquer artigo da Wikipedia. Altere o método para extrair a partir do elemento com ID `content` .

```
    public String extraiConteudoPrincipalWikipedia() {
        Element body = documento.body();
        Element content = body.getElementById("content");
        if (content != null && content.hasText()) {
            return content.text();
        } else {
            return null;
        }
    }
```

Extraindo categorias dos artigos

A maioria dos artigos da Wikipedia é categorizada. Ao pesquisar o artigo sobre *Obi-Wan Kenobi* (https://pt.wikipedia.org/wiki/Obi-Wan_Kenobi) com o inspetor de páginas, veja que as categorias estão na parte de baixo do conteúdo e se parece com isso:

```
<a href="/wiki/Categoria:Star_Wars" title="Categoria:Star
Wars">Categoria</a>
<a href="/wiki/Categoria:Personagens_de_Star_Wars"
title="Categoria:Personagens de Star Wars">Personagens de Star Wars</a>
```

Um artigo está, geralmente, associado a pelo menos uma categoria. E agora o desafio: o nome da categoria faz parte do identificador do elemento. Ainda assim existe um padrão que é `/wiki/Categoria:` . Por isso, vamos procurar os elementos da página que comecem com este padrão.

O código fica assim:

```
Elements categorias =
        content.getElementsByAttributeValueStarting(
                "href", "/wiki/Categoria:");
for (Element e : categorias) {
    // O símbolo !  indica uma categoria oculta
    if (e.text().contains("! ")) {
        continue;
    }
    lista.add(e.text());
}
```

E tem mais um detalhe. Como a Wikipedia é um site de edição coletiva, o conteúdo não segue um padrão nem tem uma curadoria muito criteriosa. A partir daí encontram-se algumas categorias significativas, como *Filmes de Pedro Almodóvar* e *História do Brasil*, nas quais conseguimos inferir o sentido. Entretanto, encontramos outras com difícil interpretação, como *1 165* e *esboços sobre geografia*.

10.3 Considerações sobre performance

Esse tipo de sistema consome muita memória e processamento do servidor. Não à toa, motivou o surgimento do *big data*, onde as soluções utilizam processamento paralelo massivo, isto é, a execução é dividida entre diversas máquinas de um *cluster*. Considerando que estamos trabalhando com apenas um computador, facilmente batemos no limite do hardware.

Os atributos de `DumpNutchV0` são do tipo `byte[]` para otimizar o uso de memória da aplicação. Sistemas que processam grandes volumes de dados, principalmente texto, consomem muita memória e devem ser feitas otimizações para economizar recursos sempre que possível. O uso de `byte[]` no lugar de String é uma dessas melhorias. Do outro lado, há um nível a mais de complexidade porque temos de converter de volta para texto para usar os valores.

Mas, sem dúvida, para processamento intensivo de dados (*data intensive*) o uso de `byte[]` no lugar de `String` é comum e até recomendado.

Considerando um *dump* de 3.4 GB, rodando os testes com o parâmetro `-Xmx8G`, o programa de teste conseguiu mostrar o conteúdo de todas as páginas em 42 segundos, usando 5.2 GB de memória. O mesmo teste demandou 7.6 GB de memória e 38 segundos quando usou o `DumpNutchVO` com Strings no lugar de `byte[]`. Perceba que o uso de memória chega a ser 46% menor e a velocidade de processamento 10% menor. Assim, rodando com `byte` é mais econômico em termo de memória e levemente mais demorado em tempo de processamento.

Veja o caso do `DumpNutchParser`, que carrega o conteúdo do *dump* na memória. Se o arquivo tiver mais de 2 GB começamos a ter problemas do tipo `java.lang.OutOfMemoryError: Java heap space` e o `java.lang.OutOfMemoryError: GC overhead limit exceeded`. Em ambos os casos, a memória é insuficiente para o programa. Neste caso, podemos aumentar o *heap* com o argumento `-Xmx` para usar mais memória.

Contudo, há outro problema, que é o `java.lang.OutOfMemoryError: Requested array size exceeds VM limit`, e desse a gente não escapa. Significa que a JVM não suporta a quantidade de itens do *array*. Aqui, a solução é reescrever o `DumpNutchParser` para suportar um maior volume de dados por meio de *threads* ou de operações em bloco.

10.4 Considerações legais e éticas

Esta seção discute os aspectos legais e éticos do web *crawling/scraping* porque, apesar de ser um livro técnico, esses outros pontos precisam ser analisados. O web *crawling* é uma

atividade que nasceu com a internet, mas isso não significa que é liberado sem restrições. Muitos sites não aceitam o rastreamento de seu conteúdo por motivos de performance, direito autoral ou simplesmente porque não querem.

Na grande maioria dos sites, a política de uso fica no rodapé. Pode ser chamada de política, condição ou termos de uso. Neste link está indicado quando o site não permite o acesso de *crawlers*. Verifique se há alguma proibição a robôs, *spiders*, *scripts*, *scrapers*, *crawlers* etc. Em um grande portal de notícias, por exemplo, encontrei uma mensagem parecida com "é proibido utilizar-se de meios automáticos, incluindo spiders, robôs, crawlers, ferramentas de captação de dados ou similares para baixar dados".

A recomendação é que, antes de apontar o *crawler* para um site, verifique as políticas de uso e a página do `robots.txt`, que é um bom lugar para aferir o que pode ou não ser vasculhado. O arquivo `robots.txt` define padrões mundialmente aceitos e o Nutch respeita essas regras. Contudo, o Nutch é um programa *open source* e pode ser alterado facilmente para ignorar as políticas do site. Assim, mesmo que um site proíba o rastreamento, você pode burlar essa orientação alterando o Nutch, o que obviamente não deve ser feito.

O web *crawling* é uma área cinza, sem muita clareza sobre os limites, o que pode ser feito e o que não pode. Mas uma coisa é certa: não podemos rastrear qualquer site. E, principalmente, nunca tente acessar dados que estão sob proteção. Mais que uma questão de legalidade jurídica, é uma questão de ética. Do ponto de vista jurídico, o problema é o uso comercial de dados de terceiros. Significa que você não pode baixar informações de portais da internet e vender sem a devida autorização, como copiar uma notícia ou um anúncio e publicar como se fosse seu. É diferente dos agregadores de notícia, onde é feita a devida citação da origem.

No Brasil há histórico de condenação pela justiça de primeira instância de uma empresa que baixou e usou dados de concorrentes, caracterizando concorrência desleal. Neste processo,

a empresa de recrutamento foi condenada a pagar uma indenização por ter baixado os currículos que estavam cadastrados em empresas concorrentes. Na dúvida, entre em contato e pergunte ao administrador do site se ele permitiria o seu rastreamento. Há casos em que o próprio portal disponibiliza os dados publicamente. E quando ele não quer, já vai estar devidamente explicitado.

O jornalismo é um grande usuário dos *crawlers* e, da mesma forma, começa a discutir questões éticas, como visto nesta reportagem (http://observatoriodaimprensa.com.br/etica-jornalistica/os-limites-da-garimpagem-de-dados-na-internet/) e nesta outra (em inglês) (http://j-source.ca/article/a-journalists-guide-to-web-scraping/). A internet, em seu estágio atual, tem um percentual considerável de conteúdo gerado automaticamente por robôs. Estes *bots* consomem conteúdo de milhares de portais, analisam seu contexto e escrevem novas notícias. Veja o caso da política mundial. Esse estudo da FGV (http://dapp.fgv.br/robos-redes-sociais-e-politica-estudo-da-fgvdapp-aponta-interferencias-ilegitimas-no-debate-publico-na-web/) indica que até 20% dos debates políticos no Brasil são originados por robôs.

A última questão a ser discutida é a quantidade de acessos. Um simples computador pessoal com 4 processadores, 16 GB de RAM e placa de rede gigabit, exatamente como o que uso para escrever esse texto, pode fazer um belo estrago em um servidor web. Nativamente, o Nutch é configurado para não gerar pouquíssimo tráfego no servidor web porque ele baixa uma página de cada vez, com intervalo de 5 segundos entre cada chamada. Essas configurações estão definidas em `fetcher.threads.per.queue` e `fetcher.server.delay`, respectivamente e tornam o rastreamento bastante lento.

Se configurarmos o `fetcher.threads.per.queue` com um valor como 100, significa que o Nutch vai tentar baixar 100 páginas simultaneamente. Caso tenha uma conexão rápida, se o site não tiver o *Crawl-Delay* configurado, enfim, se preencher várias condições, você conseguiria gerar um tráfego intenso no site

rastreado. Por outro lado, o rastreamento seria muito rápido. Isso deve ser feito em curtos períodos de tempo, caso contrário, pode ser interpretado como um Ataque de Negação de Serviço, ou *DoS Attack*. Mesmo sem considerar o ataque, você deve gerar o mínimo de tráfego possível no servidor.

Depois de todos esses argumentos, podemos ver que existe uma grande demanda para desenvolvimento de web *crawlers*, por isso escrevi esta parte do livro. Frequentemente vejo pedidos de criação de *bots* para comparação de preços de produtos, análise de portais de notícias e de editais, acompanhamento processual etc. Estas seções mostram um ponto de partida e uma direção a seguir. Bem como a direção em que não devemos seguir.

Resumo

Web *crawling* e *scraping* são técnicas para rastreamento, aquisição e extração de dados de páginas web. Isso é feito principalmente através de marcações encontradas nas páginas como título, listas, CSS etc. As ferramentas ajudam bastante neste momento, uma vez que permitem recuperar blocos específicos dentro do conteúdo da página. Outro recurso importante são as expressões regulares, para encontrar padrões como preços.

São duas fases distintas. A parte de rastreamento (*crawling*) e aquisição de dados encontra e copia o conteúdo das páginas para o disco local. Na sequência é feita a raspagem, do inglês *scraping*, dos dados, onde apenas o que é útil ao sistema é extraído, ignorando o resto do texto.

Entre as principais ferramentas estão o Nutch e o jsoup, além do próprio Lucene. Para todas elas temos exemplos completos e funcionais. Estas tecnologias podem ser usadas para fins sociais, como no caso do Jornalismo ou na área do Direito, ou para fins comerciais, como é o caso do e-commerce tradicional. Apenas deve-se observar as questões de performance, porque é possível até mesmo derrubar um site, no caso de pessoas com más

intenções. Por isso tivemos uma seção específica para tratar de questões éticas, fechando o capítulo.

Capítulo 11
Referências bibliográficas

Referências do capítulo 1

APACHE. Lista de discussão dos usuários Lucene/Solr. http://mail-archives.apache.org/mod_mbox/lucene-solr-user. Acessado em 01/03/2019.

APACHE. Lista de discussão dos desenvolvedores Lucene/Solr. http://mail-archives.apache.org/mod_mbox/lucene-dev/. Acessado em 01/03/2019.

APACHE. Bug Tracking do Lucene. https://issues.apache.org/jira/browse/LUCENE. Acessado em 01/03/2019.

APACHE. Documentação do Lucene 7. http://lucene.apache.org/core/7_4_0/index.html. Acessado em 01/03/2019.

APACHE. Wiki do Lucene. http://wiki.apache.org/lucene-java. Acessado em 01/03/2019.

APACHE. Lista de empresas que usam Lucene. http://wiki.apache.org/lucene-java/PoweredBy. Acessado em 01/03/2019.

MANNING, C.; RAGHAVAN, P.; SCHÜTZE, H. *Introduction to Information Retrieval*. http://nlp.stanford.edu/IR-book. Cambridge University Press, 2008.

Referências do capítulo 2

APACHE. Similaridade com TF/IDF. https://lucene.apache.org/core/7_4_0/core/org/apache/lucene/search

/similarities/TFIDFSimilarity.html. Acessado em 01/03/2019.

BAEZA-YATES, R; RIBEIRO-NETO, B. *Recuperação de Informação*: Conceitos e Tecnologia das Máquinas de Busca, 2ed. Bookman. 2013.

SINGHAL, A. *Modern Information Retrieval: A Brief Overview*. http://singhal.info/ieee2001.pdf/. IEEE, 2001.

Third Text REtrieval Conference (TREC 1994). http://trec.nist.gov/pubs/trec3/t3_proceedings.html/. 1994.

Referências do capítulo 3

APACHE. Lucene 7.4.0 demo API. https://lucene.apache.org/core/7_4_0/demo/overview-summary.html/. Acessado em 01/03/2019.

A simple query parser implemented with JavaCC. https://lucene.apache.org/core/7_4_0/queryparser/org/apache/lucene/queryparser/classic/package-summary.html/. Acessado em 01/03/2019.

Referências do capítulo 4

APACHE. Documentação das expressões regulares no Lucene. https://lucene.apache.org/core/7_4_0/core/org/apache/lucene/util/automaton/RegExp.html. Acessado em 01/03/2019.

JARGAS, A. M. *Expressões Regulares, uma abordagem divertida*. São Paulo: Novatec, 2016.

Regular Expressions. http://www.regular-expressions.info/. Acessado em 01/03/2019.

Referências do capítulo 5

APACHE. *Code to search indices*. http://lucene.apache.org/core/7_4_0/core/org/apache/lucene/search/

package-summary.html. Acessado em 01/03/2019.

Referências do capítulo 6

Formatos de arquivo do Lucene. http://lucene.apache.org/core/7_4_0/core/org/apache/lucene/codecs/lucene70/package-summary.html. Acessado em 01/03/2019.

Como acelerar a indexação (*How to make indexing faster*). http://wiki.apache.org/lucene-java/ImproveIndexingSpeed. Acessado em 01/03/2019.

Como acelerar a busca (*How to make searching faster*). http://wiki.apache.org/lucene-java/ImproveSearchingSpeed/.

Java VisualVM - Profiling Applications. http://docs.oracle.com/javase/7/docs/technotes/guides/visualvm/profiler.html/. Acessado em 01/03/2019.

Teste de performance do Lucene. http://people.apache.org/~mikemccand/lucenebench/indexing.html. Acessado em 01/03/2019.

Utilitários para o teste de performance do Lucene. https://github.com/mikemccand/luceneutil/. Acessado em 01/03/2019.

Referências do capítulo 7

How Twitter Uses Apache Lucene for Real-Time Search. https://lucidworks.com/blog/2015/09/02/how-twitter-uses-apache-lucene-for-real-time-search/. Acessado em 01/03/2019.

Near-real-time readers with Lucene's SearcherManager and NRTManager. http://blog.mikemccandless.com/2011/11/near-real-time-readers-with-lucenes.html/. Acessado em 01/03/2019.

Referências do capítulo 8

Design a Fluent API in Java. https://dzone.com/articles/java-fluent-api-design/. Acessado em 01/03/2019.

Hibernate Search 5.11.1.Final: Reference Guide. https://docs.jboss.org/hibernate/stable/search/reference/en-US/html_single/. Acessado em 01/03/2019.

The Trie Data Structure: A Neglected Gem. https://www.toptal.com/java/the-trie-a-neglected-data-structure/. Acessado em 01/03/2019.

Referências do capítulo 9

At a loss for words? . https://googleblog.blogspot.com.br/2008/08/at-loss-for-words.html/. Acessado em 01/03/2019.

Referências do capítulo 10

A journalist's guide to web scraping. http://j-source.ca/article/a-journalists-guide-to-web-scraping. Acessado em 01/03/2019.

Alexa: principais sites pela categoria. https://www.alexa.com/topsites/category. Acessado em 01/03/2019.

Associação Brasileira de Lawtechs e Legaltechs. AB2L. https://www.ab2l.org.br. Acessado em 01/03/2019.

BRIN, S; PAGE, L. *The Anatomy of a Large-Scale Hypertextual Web Search Engine*. http://infolab.stanford.edu/~backrub/google.html. Acessado em 01/03/2019.

CSS básico. https://developer.mozilla.org/pt-BR/docs/Aprender/Getting_started_with_the_web/CSS_basico. Acessado em 01/03/2019.

FGV, DAPP. ROBÔS, REDES SOCIAIS E POLÍTICA. http://dapp.fgv.br/robos-redes-sociais-e-politica-estudo-da-fgvdapp-aponta-interferencias-ilegitimas-no-debate-publico-na-web. Acessado em 01/03/2019.

HEDLEY, J. Jsoup Cookbook. https://jsoup.org/cookbook. Acessado em 01/03/2019.

Introdução ao DOM. https://developer.mozilla.org/pt-PT/docs/Gecko_DOM_Reference/Introduction. Acessado em 01/03/2019.

MITCHELL, R. *Web Scraping with Python*: Collection Data from the Modern Web. O'Reilly, 2015.

Nutch 1.x REST API v1.0. https://wiki.apache.org/nutch/Nutch_1.X_RESTAPI/. Acessado em 01/03/2019.

Page Inspector do Firefox. https://developer.mozilla.org/en-US/docs/Tools/Page_Inspector/UI_Tour. Acessado em 01/03/2019.

REIS, Marco. *Lista de datasets para download*. http://blog.marcoreis.net/lista-de-datasets-para-download. Acessado em 01/03/2019.

SHIAB, N. *Os limites da garimpagem de dados na internet*. Observatório da Imprensa. http://observatoriodaimprensa.com.br/etica-jornalistica/os-limites-da-garimpagem-de-dados-na-internet. Acessado em 01/03/2019.

Web Semântica. http://www.w3c.br/Padroes/WebSemantica. Acessado em 01/03/2019.