Busca parametrizada para a recuperação ranqueada na Web

Fábio Eduardo Kaspar
Igor Canko Minnoto
Ricardo Oliveira Teles

Trabalho de formatura supervisionado
Instituto de Matemática e Estatística
da
Universidade de São Paulo

Orientador: Prof. Dr. João Eduardo Fereira
Coorientador: André Casimiro

São Paulo, novembro de 2015

Busca parametrizada para a recuperação ranqueada na Web

Esta é a versão original do trabalho de formatura
supervisionado elaborado pelos alunos Fábio Eduardo Kaspar,
Igor Canko Minnoto e Ricardo Oliveira Teles.

Resumo

Com o grande volume de dados ao qual temos acesso na Internet, é necessário o uso de ferramentas de busca para encontrar informações específicas em coleções de páginas Web. Muitas dessas ferramentas utilizaram sistemas de Recuperação de Informação (RI), pois permitem consultas eficientes utilizando estruturas de dados construídas com antecedência. Nesse contexto, alguns sites surgiram com a proposta de facilitar a descoberta de estabelecimentos, como bares e restaurantes. Utilizando consultas puramente textuais ou com o auxílio de parâmetros estruturados, esses sistemas se popularizaram entre os usuários. Contudo, a forma de construir essas consultas pode gerar frustrações caso não os permita comunicar suas necessidades. Com o objetivo de estudar uma outra possibilidade de formulação de consultas, foi desenvolvido a aplicação *LookingFor*. Nele, o usuário não fornece valores para os parâmetros, mas organiza-os em uma ordem de prioridade. Dessa forma, não é necessário ler as listas de opções antes de construir uma consulta, mas apenas ordenar os poucos parâmetros disponíveis. Foi realizada uma entrevista com um grupo de 19 usuários, para que relatassem sua experiência com o sistema. A partir desses relatos, pode-se constatar algumas necessidades que não foram contempladas por essa aplicação.

Palavras-chave: Recuperação de Informação, busca parametrizada, parâmetros, pesos.

Abstract

With the big volume of data that Internet provides us, it is essential to use search engines to find particular information in collections of Web pages. Many of the engines use Information Retrieval (IR) systems, for they allow efficient searches with previously built data structures. In this context, some sites emerged proposing to facilitate the discovery of establishments, like bars and restaurants. Using purely textual queries or with the help of structured parameters, these systems became popular among users. However, the method of formulating queries can be frustrating in case it does not grant ways to comunicate user's needs. With the goal of studying another possibility of formulating queries, we have developed the *LookingFor* site. The user does not provide values to parameters, but organizes them according to priority. In this way, reading options lists before constructing a query is not necessary, being sufficient to just order a few parameters available. An interview was applied with 19 users, so they could report their experience about the system. From these reports, it was possible to find some needs that weren't contemplated by this aplication.

Keywords: Information Retrieval, parametric search, parameters, weights.

Sumário

1 Introdução **1**

2 Fundamentos **4**

2.1 Recuperação booleana 4

2.2 Vocabulário de termos e listas de postagens 6

2.2.1 Decodificação de sequências de caracteres e delimitação de documento 6

2.2.2 Escolhendo a unidade de documento 6

2.2.3 Determinando o vocabulário de termos 6

2.2.4 Intersecção de listas de postagens mais rápida via *skip pointers* 7

2.2.5 Postagens posicionais e consultas de frase 7

2.3 Dicionários e recuperação tolerante 8

2.4 Contrução do índice 9

2.5 Pontuação, ponderação do termo e modelo do espaço vetorial 11

2.5.1 Atribuir pontuação a um documento 11

2.5.2 Índices paramétricos e de zonas 12

3 Desenvolvimento **14**

3.1 Coleta de documentos e metadados 15

3.2 Indexação 15

3.3 Busca parametrizada 16

3.4 Interface de usuário e georreferenciamento 17

4 Aplicação **19**

4.1 Resultados 19

4.1.1 Consultas 20

4.1.2 Estabelecimentos escolhidos 21

4.2 Feedback dos usuários 22

5 Conclusões **24**

A Formulário **25**

Capítulo 1

Introdução

Do mundo real ao problema computacional

São Paulo é a segunda maior cidade do mundo em número de restaurantes. Conhecida como a Capital Latino-Americana de boa mesa, possui mais de 15 mil restaurantes, 500 churrascarias, 4.500 pizzarias, 20 mil bares, entre outros [Vis]. Dessa forma, encontrar um estabelecimento gastronômico dentre mais de 40 mil opções é uma tarefa impraticável. Surge então a demanda por formas de encontrar uma opção desejada sem métodos exaustivos e de forma eficiente. Para suprir essa demanda, muitas ferramentas de busca atuais utilizam sistemas de Recuperação de Informação (RI).

Segundo Manning, Raghavan e Schütze (2008), Recuperação de Informação é encontrar material (geralmente documentos) de uma natureza não estruturada (geralmente textos) que satisfaça uma informação necessária dentro de grandes coleções (geralmente armazenadas em computadores). Essa área está se tornando rapidamente a forma mais utilizada de acesso a informação, pois permite a recuperação rápida de referências a determinados termos. Isso possibilita aos usuários fazerem buscas de forma não estruturada, utilizando também a linguagem natural. Os resultados são ordenados em alguma ordem de relevância, fazendo emergir resultados que possam ser mais interessantes.

Um dos problemas dessa área é determinar essa ordem em que os resultados devem ser apresentados. Os autores definem a relevância dos resultados como a percepção do usuário sobre as informações contidas no documento e se elas vão ao encontro à sua necessidade. Portanto, a relevância é subjetiva ao usuário. Para Crippa e Rodrigues (2011), é justamente essa subjetividade que dificulta alcançar o objetivo da área de fornecer acesso rápido e eficaz às informações relevantes.

A área de RI utiliza estatísticas relacionados a termos e documentos para tentar quantificar essa relevância. Manning, Raghavan e Schütze (2008) dão alguns exemplos de dados que permitem comparar documentos, como a frequência dos termos em um documento, a raridade de um termo na coleção e a quantidade de acessos a uma página Web.

Existem muitos sites na Web que utilizam sistemas de RI, tendo cada um deles formas diferentes de criar consultas e apresentar resultados. Escudeiro e Jorge (2008) definem três abordagens utilizadas por esses sistemas:

1. *Search-centric approach*: advoga que a busca textual livre se tornou tão boa e seu estilo de interface, tão comum, que os usuários podem satisfazer todas as suas necessidades por meio de consultas simples. Algumas ferramentas de busca utilizam essa abordagem.

2. *Taxonomy navigation approach*: afirma que os usuários têm dificuldades em expressar a informação necessária. Para ajudar a encontrá-la, essa abordagem propõe o uso de estruturas hierárquicas para organizar informações. Um exemplo dessa abordagem são os sistemas de buscas em diretórios.

3. *Meta-data centric approach*: com o auxílio de metadados, podemos filtrar grandes conjuntos de resultados. Algumas ferramentas de busca estão tentando melhorar a qualidade de suas respostas usando evidências diversas, seguindo essa abordagem.

Como exemplo do primeiro modelo, podemos citar o site Google [Gooa]. Sua busca fornece apenas um campo de texto. Qualquer especificação da busca deve ser feita textualmente. Na sua variante GoogleMaps [Goob], o mesmo modelo é utilizado. Nesse site é possível buscar por estabelecimentos como bares e restaurantes. Após a busca ser feita, o usuário tem apenas a opção de filtrar os resultados por uma "classificação" que é fornecida pela comunidade.

Este Trabalho de Conclusão de Curso (TCC), por sua vez, se refere ao terceiro modelo, o *metadata centric approach*. Esse modelo nos orienta a buscar evidências de informações necessárias em outras partes do documento, como o título e os metadados.

Por exemplo, buscando por sites de restaurantes, um usuário pode achar mais relevantes os resultados com preços mais baixos ou que são mais próximos de sua residência. Pode também querer apenas os restaurantes de melhor qualidade, sem se preocupar com a distância e preço.

Para satisfazer essa necessidade do usuário, não é suficiente olhar para o corpo do documento. Nos metadados, podemos guardar o preço médio das refeições e a coordenada geográfica do estabelecimento. Utilizamos essas informações para filtrar e ranquear os resultados. Nesse modelo, portanto, o usuário tem mais liberdade para comunicar sua necessidade e atuar sobre o ranqueamento dos resultados, contrapondo o primeiro modelo.

A forma como essa busca é feita varia com a interface que é apresentada para o usuário. Sites como o GuiaFolha [Gui] e o Kekanto [Kek] utilizam um esquema de caixas de seleção para ajudar o usuário a encontrar os resultados mais relevantes. Como podemos ver na figura 1.1, existem muitas opções disponíveis que, mesmo agrupadas, podem formar diversas combinações.

Figura 1.1: *Interface do site GuiaFolha [Gui]*

Por exemplo, é possível filtrar os restaurantes com gasto médio entre R$75 e R$95 e que se situam no bairro do Butantã. Não é possível, no entanto, ordenar os resultados por proximidade e preços, balanceando os dois parâmetros de alguma forma. Se o usuário não tem certeza de que filtros usar, precisa testar configurações diferentes, possivelmente realizando várias buscas até encontrar um resultado satisfatório.

Assim, este trabalho tem como principal objetivo desenvolver uma alternativa de busca baseada

em metadados de modo a viabilizar a implementação de uma aplicação web que auxilie os usuários a encontrar restaurantes de acordo com alguns poucos critérios, visando simplificar buscas que utilizam metadados.

Capítulo 2

Fundamentos

2.1 Recuperação booleana

Com o passar do tempo os sistemas de busca foram se popularizando e aprimorando, com respostas que facilitam a tomada de decisão do usuário, fornecendo um conjunto de opções supostamente mais relevantes. Esse processo é conhecido como Recuperação de Informação.

"Recuperação de Informação é encontrar materiais (normalmente documentos) de natureza não-estruturada (normalmente texto) que satisfaçam uma informação necessária em grandes coleções (normalmente armazenadas em computadores)." (Manning, Raghavan, Schütze, 2008, p.1)

Por décadas, utilizou-se sistemas de recuperação de informação em ambientes restritos com usuários específicos utilizando alguma linguagem de consulta. No começo da década de 90, coincidentemente com o advento da Internet, sistemas com consultas em linguagem natural tiveram maior destaque.

Não seria prático executar tais buscas em todos os documentos toda vez que uma informação fosse requerida, tornando-se um processo inviável dependendo do tamanho da coleção. Por exemplo, com o número de páginas na Internet esse processo seria extremamente lento. Portanto, criar um índice dos documentos de antemão permite executar a busca mais rapidamente, ao custo do armazenamento e manutenção dele.

Uma forma de fazer tais buscas é por meio do Modelo de Recuperação Booleana (do inglês, *Boolean Retrieval Model*), o qual permite consultas com expressões booleanas, utilizando termos e operadores lógicos como *AND*, *OR* e *NOT* para conectá-los. Com esse modelo, o usuário consegue encontrar precisamente que documentos dentro da coleção satisfazem a sua busca.

Cada documento possui um conjunto de termos nele presente, que o distingue de outros, e para cada termo que se busca, existem documentos em que esse está presente. Assim, a estrutura de dados mais intuitiva a priori é a matriz de incidência, na qual cada posição pode ser 1 ou 0, se o documento contém o termo ou não. Portanto, essa matriz é criada levando-se em conta todos os termos presentes na coleção. Para saber quais documentos contêm certo termo, consulta-se a linha da matriz correspondente ao termo.

Voltando ao índice, para uma determinada coleção de documentos, o conjunto de termos presentes é chamado de vocabulário ou léxico e a estrutura de dados que guarda o vocabulário é chamada de dicionário.

Existe aqui uma diferença entre a informação necessária, que é aquilo que o usuário visa encontrar de fato, e consulta, que é o que o usuário fornece ao sistema. Um documento é dito relevante se contém informações que o usuário julgue compatíveis com o que lhe é necessário. Podemos medir a eficiência do sistema quanto a uma busca de duas formas: sua precisão ou quantos dos documentos retornados são relevantes; e seu *recall* ou que parcela dos documentos relevantes do sistema foi retornada.

Tendo uma consulta de dois termos "t_1 *AND* t_2", o modelo de recuperação booleana considera

as linhas da matriz de incidência como dois números binários e aplica o operador. Portanto nesse modelo, temos um baixo *recall*, pois o sistema filtra os documentos a partir da consulta, não tendo flexibilidade para incluir outros que poderiam ser relevantes. É um modelo muito limitado, tendo em vista as necessidades dos usuários.

Em geral, a matriz de incidência dos termos é extremamente esparsa. Se tivermos documentos de 1000 palavras, mas um dicionário com 10^6 termos, teremos cada linha da matriz com no máximo 0,1% das posições com 1.

Entretanto, existe uma estrutura mais compacta chamada Lista Invertida (do inglês, *inverted index*), que indica para cada termo do dicionário os documentos em que esse está presente. Cada registro nessa lista é chamado de postagem (do inglês, *posting*). Para cada termo temos uma lista de postagens. Para gerar o índice temos os seguintes passos:

1. Reunir os documentos que farão parte do índice.
2. Criar *tokens* para as palavras criando uma lista para cada documento.
3. Normalizar estes *tokens* por meio de um processo linguístico. Os *tokens* normalizados serão os termos indexados (o dicionário).
4. Atribuir uma identificação (por exemplo, número inteiro em série) para cada documento e atribuí-los às listas dos termos que neles aparecem.

Depois, ordena-se o dicionário em ordem alfabética. Ocorrências repetidas de um mesmo termo em um documento são mescladas e entradas repetidas de um mesmo termo no dicionário são agrupadas. O produto final é composto pelo dicionário de termos e suas postagens. O dicionário pode ser armazenado na memória, dependendo de seu tamanho, enquanto as postagens serão lidas do disco.

Essa estrutura de dados pode ser usada para guardar estatísticas, como o número de documentos em que um termo aparece (que é também o comprimento da lista de postagens). Essas estatísticas podem ser usadas para ranquear os resultados de uma busca de forma mais eficiente.

As listas de postagens podem ser feitas com diferentes estruturas de dados, como vetores e listas ligadas. Vetores de tamanho fixo são pouco eficientes, pois podemos ter listas de postagens de tamanhos muito discrepantes, o que resultaria em desperdício de espaço. Com vetores de tamanho variável, perde-se tempo apenas no redimensionamento, logo, se o índice não for muito atualizado, esse pode ser útil. Por sua vez, listas ligadas precisam de mais espaço por causa dos ponteiros.

No modelo de recuperação booleana, tendo essas listas ordenadas pelas identificações dos documentos, é fácil de processar buscas. No caso do operador *AND*, basta recuperar as listas de dois dos termos e selecionar os termos presentes nas duas listas. Já que as listas estão ordenadas podemos avançar intercaladamente entre elas. Sendo n_i o tamanho de cada uma das listas, essa operação tem $O(n_1 + n_2)$ comparações. Para processar a consulta inteira, pode-se fazer uma operação de cada vez. Assim, a complexidade da busca para uma consulta genérica é $\Theta(N)$, sendo N o tamanho do dicionário, que na prática é uma constante imensa.

Uma forma de otimizar o processamento da consulta é mudar a ordem em que as operações são feitas, ordenando os termos da consulta pelo tamanho de suas listas de postagens. Nenhum resultado parcial será maior do que a menor das listas utilizadas até ali, logo, começando pelas operações com as listas menores, o tamanho dos resultados parciais será sempre menor ou igual a menor das listas utilizadas na consulta.

Existem outros modelos de recuperação de informação, como modelos de recuperação ranqueada (do inglês, *ranked retrieval models*), no qual a consulta submetida pelo usuário assume formato livre e o sistema deve responder a essas consultas. Um exemplo é o modelo de espaço vetorial contrastando com o uso de operadores.

2.2 Vocabulário de termos e listas de postagens

2.2.1 Decodificação de sequências de caracteres e delimitação de documento

Os documentos digitais são entradas para um indexador e são tipicamente bytes de um arquivo ou de uma página web e, sendo assim, a primeira etapa do processo de indexação é converter esses bytes para uma sequência linear de caracteres. Mas antes é preciso determinar a codificação correta do documento. Determinada a codificação, é feita a conversão de bytes para caracteres.

2.2.2 Escolhendo a unidade de documento

Para uma coleção de livros, normalmente seria má ideia indexar um livro inteiro como um documento, pois a busca por "brinquedos chineses" pode retornar um livro que menciona China no primeiro capítulo e brinquedos no último, o que não torna o livro relevante. Ao invés disso, podemos indexar cada capítulo ou parágrafo ou até mesmo sentenças individuais como documentos. Com isso, uma vez que os documentos são menores, será muito mais fácil para o usuário encontrar passagens relevantes dentro do documento. Torna-se claro que há uma recompensa com isso em termos de *precision/recall*. Se as unidades forem muito pequenas, é provável que se perca passagens importantes, porque os termos foram distribuídos sobre vários mini-documentos, enquanto que se as unidades forem muito grandes, há a tendência de obtermos correspondências espúrias e a informação relevante torna-se difícil de ser encontrada pelo usuário.

2.2.3 Determinando o vocabulário de termos

Tokenização

Tokenização é a tarefa de dividir uma sequência de caracteres em pedaços menores, chamados de *tokens*, podendo eventualmente descartar certos caracteres, como pontuação. Um *token* é uma instância de cadeia de caracteres em um documento que é agrupada como uma unidade semântica útil para processamento. Segue abaixo um exemplo de tokenização:

Entrada: Friends, Romans, Countrymen, lend me your ears

Saída: [Friends] [Romans] [Countrymen] [lend] [me] [your] [ears]

Um tipo é a classe de todos os *tokens* contendo a mesma sequência de caracteres. Um termo é um tipo (talvez normalizado) que é incluso no dicionário do sistema de RI. Múltiplos *tokens* que são reunidos em conjunto via normalização são indexados como um termo sob forma normalizada. Por exemplo, se o documento a ser indexado é *"to sleep perchance to dream"*, então há 5 *tokens*, mas apenas 4 tipos (uma vez que há 2 instâncias de *to*). Entretanto, se *to* for omitido do índice (como uma *stop word*), então haverá somente 3 termos: *sleep*, *perchance*, e *dream*.

A principal questão na fase de tokenização é qual convenção usar? Corta-se nos espaços em branco ou não, remove-se os caracteres de pontuação, o que fazer em casos com hífen, etc. Cada idioma apresenta questões particulares que devem ser consideradas e alguns até mesmo fazem da tokenização uma tarefa muito complexa. Todos os métodos cometem erros algumas vezes, e então nunca há a garantia de uma tokenização única e consistente.

Eliminando termos comuns: *stop words*

Stop words são *tokens* que não são indexados e, portanto, não são termos do vocabulário. Tais *tokens* são muito comuns e parecem ser de pouco valor para tornar um documento relevante ao usuário (e.g. artigos, pronomes, preposições, entre outros).

Alguns sistemas de RI excluem do vocabulário as *stop words* e para isso, determina-se os termos mais frequentes da coleção, podendo ser necessário estudar a relação da semântica do termo com o domínio de indexação. Entretanto, nem sempre é razoável eliminar as *stop words*, pois podem ser úteis em casos de consultas de frases (e.g. "presidente do Brasil" é mais preciso do que "presidente" *AND* "Brasil").

A princípio, o custo de inclusão de *stop words* não é tão grande, tanto em armazenamento quanto em processamento, e em geral, ferramentas de busca da Web nem as utilizam.

Normalização (classes de equivalência de termos)

Nesta etapa é que são gerados os termos que aparecerão no dicionário. Após "quebrar" a coleção e a consulta em *tokens*, casos de correspondência exata entre os *tokens* não são sempre verdade, mas há casos onde é desejável agrupar *tokens* semelhantes por classes de equivalência ou por algum outro método de agrupamento.

Agrupá-los por classes de equivalência tem suas vantagens em termos de desempenho e significa aplicar regras de eliminação de hífens, diacríticos, redução para o minúsculo (e entre outros) e o resultado dessas regras geram o nome das classes. Em outras palavras, implica converter os *tokens* para a forma canônica. Outras questões particulares de cada idioma devem ser levadas em conta tanto na etapa de tokenização quanto de normalização e muitas vezes é útil a aplicação de classificadores de idioma para a melhor seleção das regras em ambas as etapas.

Stemização e Lematização

Ambos termos se referem a algoritmos que permitem a redução de palavras flexionadas ou derivadas à sua forma base ou raiz, permitindo que variantes gramaticais de uma palavra sejam agrupadas como uma só unidade. A normalização continua sendo o objetivo aqui.

Stemização geralmente se refere a uma heurística bruta de cortes de afixos (isto é, prefixos e sufixos) para encontrar a raiz da palavra, o que torna sua implementação mais fácil, rápida e satisfatória para muitas aplicações, permitindo alto *recall*, mas baixa precisão.

Lematização faz uso de vocabulário e análise morfológica para identificar a classe gramatical da palavra, e assim determinar o lema da palavra, que é a sua forma de dicionário (forma raiz). Diferente da stemização, a lematização distingue o contexto das palavras, selecionando seu radical apropriado.

Exemplo:

1. A palavra "*better*" possui "*good*" como lema. Essa associação é perdida na stemização, mas obtida na lematização, pois requer um dicionário.

2. A palavra "*walk*" é a forma base da palavra "*walking*", e portanto isso é conseguido tanto pela stemização como pela lematização.

2.2.4 Intersecção de listas de postagens mais rápida via *skip pointers*

Se o índice for relativamente estático, é possível implementar intersecção de listas de postagens mais eficiente usando *skip pointers*. Tradicionalmente, dadas duas listas de tamanho m e n, a operação de intersecção consome tempo $O(m+n)$, pois percorre simultaneamente ambas as listas. Já com *skip pointers*, é possível reduzir essa complexidade para tempo sublinear. Alocando-se uma quantidade arbitrária e heuristicamente razoável de ponteiros, ganha-se tempo ao evitar ter de comparar postagens menores de uma das listas com o elemento de comparação da outra lista, pois não apareceriam de qualquer maneira no resultado da operação.

2.2.5 Postagens posicionais e consultas de frase

Na maioria das situações, os usuários irão expressar sua consultas por meio de frases, e assim é desejável que a ferramenta de busca forneça suporte adequado e eficiente para consultas mais complexas que envolvam múltiplos termos, onde a proximidade entre eles é um fator de relevância adicional.

Índice de dupla palavra

Após a tokenização, é necessário realizar classificação linguística das palavras (e.g. susbtantivo, verbo, preposição, entre outros) por meio do uso de ferramentas computacionais próprias. Com isso, torna-se possível indexar os documentos como duplas palavras estendidas na forma N X* N, onde N representa uma palavra "relevante" e X, uma palavra funcional de baixa relevância, como preposição, artigo ou conjunção. Assim, as entradas do dicionário passam a assumir a forma N X* N, tornando possível a realização de consultas de frase por parte do usuário final. Tal conceito pode ser estendido para mais de duas palavras, e assim denominando-se índice de frases. Índices de frases em geral expandem muito o vocabulário, e não impedem que falsos positivos ocorram. Quando o índice de dupla palavra é usado, deve-se manter também um índice de termos únicos, para que buscas por termos individuais possam ser realizadas.

Exemplo de formato de dupla palavra:

Tabela 2.1: *Formato de dupla palavra.*

renegotiation	*of*	*the*	*constitution*
N	X	X	N

Índices posicionais

Índices de dupla palavra não são o padrão, por razões de espaço requerido. O mais usado para dar suporte a consultas de frase e de proximidade é o índice posicional:

to , 993427: < 1, 6: < 7, 18, 33, 72, 86, 231 > ; 2, 5: < 1, 17, 74, 222, 255 >; ...>

No exemplo acima, o termo to tem valor df 993427 (nº de ocorrências do termo na coleção). A primeira postagem são ocorrências do termo no documento 1, e seu valor tf vale 6 (número de ocorrências do termo no documento). A lista em seguida são as posições de ocorrência do termo no documento. Porém, os índices posicionais tendem a exigir mais requisitos de espaço e a ser menos eficiente do que os outros índices.

2.3 Dicionários e recuperação tolerante

Hashes e árvores são estruturas de dados extremamente importantes na realização de consulta, pois ajudam a determinar se cada termo da consulta existe no vocabulário. No primeiro, cada termo do vocabulário(chave) é mapeado para um inteiro com espaço suficiente para que as colisões de *hash* sejam improváveis. No segundo, a árvore de busca mais conhecida é a árvore binária, na qual cada nó interno tem dois filhos. A busca de um termo começa na raiz da árvore. Cada nó interno (incluindo a raiz) representa um teste binário, baseado nesse resultado a busca prossegue para uma das sub-árvores abaixo desse nó. Contudo, um dos problemas enfrentados pelas árvores binárias é o rebalanceamento, portanto, para suavizar esse problema são usadas árvores B, as quais são árvores de busca em que cada nó interno tem um número de filhos no intervalo [a-b].

Um ponto negativo dos *hashes* é que em uma ferramente (tal como a Web) cujo tamanho do vocabulário permanece aumentando, uma função de *hash* projetada para necessidade atual pode não ser suficiente daqui algum tempo.

Há consultas conhecidas como "consultas curingas", as quais são, normalmente, usadas pelo usuário quando ele não tem certeza de como se escreve um termo da consulta ou está ciente das várias formas de escrita de um termo, então procura documentos contendo qualquer uma dessas variações. Para realizar essas consultas curingas, o usuário precisa digitar o caracter *, como nos exemplos abaixo:

S*dney → Sydney ou Sidney
*artoze → Catorze ou Quatorze

Para manipular consultas em que há um único símbolo *, tal como "S*dney", costuma-se usar duas árvores-B, árvore-B normal e árvore-B reversa, e depois pegar a intersecção de ambas.

Além disso, algumas técnicas robustas para consultas curingas e correções ortográficas são utilizadas tais como os índices permuterm e os índices k-grama.

O índice permuterm consiste em marcar o final do termo com o caracter $ e então buscar no dicionário todas as rotações com esse índice. Um exemplo do índice permuterm para a palavra carro é: carro$, arro$c, rroca, rocar, o$carr. Uma desvantagem do índice permuterm é que o seu dicionário se torna bastante grande, já que inclui todas as rotações de cada termo.

Dessa forma, uma outra técnica seria o índice k-grama. Um k-grama é uma sequência de k caracteres. Além disso, usa-se um caractere especial $ para denotar o começo ou o fim de um termo, de modo que o conjunto completo de 3-gramas gerado para o termo carro é: ca, arr, rro, ro. Em um índice k-grama, o dicionário contém todos os k-gramas que ocorrem em qualquer termo do vocabulário. Cada lista de postagem aponta de um k-grama para todos os termos de vocabulário que contém esse k-grama.

2.4 Contrução do índice

Para criar uma lista invertida para uma coleção de documentos, primeiro cria-se um lista de pares termo-*docID*, ordenando-a pelos termos e mantendo a sequência da identificação dos documentos. Com isso, podemos criar estatísticas como a frequência de documentos e a do termo. Para coleções pequenas, a memória pode ser suficiente para esta tarefa. Para coleções maiores, no entanto, será necessário também o uso do disco.

Para a discussão que se segue, vale lembrar alguns pontos:

- O acesso à memória é muito mais rápido do que acessar dados no disco.
- A movimentação da *disk head* é em geral muito lenta, e durante esse processo não há transferência de dados. Logo, é melhor transferir dados que estejam agrupados no menor número de blocos possível.
- Os sistemas operacionais leem blocos inteiros do disco, logo, ler 1 byte ou um bloco inteiro representa a mesma operação.
- A leitura do disco não é executado pelo processador, logo, esse está livre durante a leitura. Uma forma de se aproveitar esse fato é gravar dados comprimidos no disco e ter um algoritmo de descompressão eficiente, de forma que a leitura mais o tempo de descompressão dos dados seja menor que a leitura dos dados descomprimidos.

Operações com strings podem ser custosas, portanto, é preferível o uso de *termIDs* ao invés dos termos em si. Estas identificações podem ser geradas durante a criação da lista invertida ou em uma etapa anterior, criando primeiramente o dicionário.

Se cada *termIDs* ou *docIDs* ocupar 4 bytes, uma coleção de 100 milhões de tokens somaria 0,8 GB. A ordenação destes pares tomará ainda mais espaço, dependendo do algoritmo que for utilizado. A memória sendo insuficiente, torna-se necessário o uso do disco. Além disso, o uso de algoritmos de ordenação externa que minimizem o número de acessos aleatórios ao disco, já que buscas em blocos sequenciais são mais rápidas. Um exemplo desse tipo de algoritmo é o *blocked sort-based indexing algorithm* (BSBI).

Esse algoritmo cria blocos de mesmo tamanho contendo os pares *termID-docID*, ordena cada bloco em memória guardando os resultados parciais no disco e, por fim, mescla os blocos na lista final. Os blocos devem caber na memória de forma que permitam a sua ordenação sem o uso do disco. Após a ordenação, cada bloco é uma lista invertida de um pedaço da coleção. Assim, para

mesclar as listas, mantém-se aberto um buffer de leitura para cada bloco e um buffer de escrita, onde será escrito a lista resultante. Escolhemos o menor *termID* que não está na lista final e mesclamos as listas de postagens de todos os blocos; essa será a lista de postagens desse *termID* na lista inversa resultante.

A complexidade do algoritmo é equivalente à uma ordenação. Sendo T um limitante superior proporcional a quantidade de pares *termID-docID*, temos que esta complexidade é $\Theta(T \cdot logT)$. Na prática, os passos de análise dos documentos e mesclagem da lista final influenciam o tempo total da execução do algoritmo.

O algoritmo BSBI requer que a estrutura de dados que mapeia termos aos seus respectivos *termIDs* seja guardada na memória. Para coleções muito grandes isso se torna impraticável. Uma alternativa é o algoritmo *single-pass in-memory indexing* (SPIMI) que usa os próprios termos ao invés de seus *termIDs*, criando um dicionário para cada bloco e guardando-o no disco.

Primeiro, é necessário parsear os documentos em pares de termos e *docIDs*. Para cada um destes pares, verifica-se no dicionário se o termo já está presente e resgata-se a sua lista de postagens. Assim, cada postagem é adicionada a lista de postagens individualmente, diferentemente do algoritmo BSBI que primeiro recolhe todos os pares para depois organizá-los. Isso torna o algoritmo SPIMI mais rápido por não precisar fazer este último passo e, além disso, não precisa guardar os *termIDs* por manter uma relação entre um termo e a sua lista de postagens, o que permite utilizar melhor a memória com blocos maiores. Quando a memória estiver cheia, escreve-se a lista em bloco no disco, apenas tomando o cuidado de ordenar os termos do dicionário, o que facilita a mesclagem das listas, por permitir encontrar um termo com uma busca linear em qualquer dicionário. Por fim, mescla-se as listas de todos os blocos.

O algoritmo SPIMI também permite a compressão das listas de postagens e do dicionário, o que permite processar blocos maiores de cada vez e economiza espaço do disco. Por não precisarmos ordenar uma lista de postagens, o algoritmo tem complexidade $\Theta(T)$, ou seja, não tem operações que sejam mais do que lineares no tamanho da coleção. Mas, e se uma coleção for muito grande para ser indexada em um único computador? Neste caso, são utilizados algoritmos de indexação distribuída, que criam um índice armazenado em mais de uma máquina. Cada máquina pode conter um pedaço de cada lista de postagens ou, para um conjunto de termos, suas listas de postagens. Assim, podemos usar o processamento distribuído de um *cluster*, distribuindo parcelas da coleção de documentos a ser indexada para cada máquina.

Um problema com esse método é a atribuição de *termIDs* para os termos, já que cada unidade de processamento terá uma tabela própria. Um modo de contornar isso é preprocessar uma tabela de *termIDs* para os termos mais recorrentes que será distribuída para todas as máquinas e, para os termos menos frequentes, atribuir as listas de postagens aos termos em si. Com isso, cada máquina cria arquivos intermediários que guardam um conjunto de pares de valores para as listas de postagens. Por exemplo, para cada máquina teríamos um arquivos com os pares correspondentes aos termos que começam com a letra 'a' até a letra 'g', outro arquivos para os termos que começam com a letra 'h' até a letra 'p', e assim por diante. Depois estes arquivos serão mesclados de forma que uma máquina contenha todos os pares para um destes segmentos de termos. Digamos, uma máquina recebe os arquivos que contém os pares com os termos de 'a' até 'g', mescla estas listas e mantém as listas de postagens resultantes.

Tanto os algoritmos BSBI e SPIMI quanto a indexação distribuída trabalham, a princípio, com coleções estáticas, ou seja, cada documento será indexado apenas uma vez. Mas coleções que se modificam ao longo do tempo precisarão de atualizações eventualmente. Se as mudanças forem pouco frequentes, uma opção seria reindexar a coleção inteira, mantendo uma versão estável para as buscas. Se for necessário adicionar versões mais recentes dos documentos de forma rápida, no entanto, isto deixa de ser uma opção.

Uma lista auxiliar seria útil neste caso. Manteríamos a lista invertida original em disco e a lista auxiliar com as entradas novas, em memória. Ao invés de atualizarmos frequentemente a lista, podemos concentrar as mudanças nesta lista auxiliar e fazer acessos ao disco de uma vez só. Com este esquema, uma busca percorre ambas as listas para obter seu resultado. Entradas

novas são adicionadas a lista auxiliar e entradas que foram deletadas são mantidas em um vetor de validação ou estrutura similar que possa ser usada para filtrar os resultados. Assim, para atualizar um documento, basta deletar suas entradas da lista original (ou seja, adicioná-las ao vetor de deleções) e adicionar as novas entradas na lista auxiliar. Quando a lista auxiliar estiver muito grande, enfim, mesclamos as duas listas. Se tivéssemos cada lista de postagens em um arquivo separado isto seria fácil, bastaríamos mesclar as mudanças pertinentes a este arquivo. Mas em geral este não é o caso, já que os sistemas operacionais não lidam bem com um número muito grande de arquivos.

Para que o número de arquivos não cresça demasiadamente, uma possibilidade é usar o algoritmo *logarithmic merge*, que utiliza um conjunto de listas guardadas em memória cada uma com o dobro do tamanho da anterior. O tamanho da primeira lista é igual ao tamanho máximo n da lista auxiliar e toda vez que um nível estiver cheio, mesclamos com o próximo. Para um conjunto de postagens com T elementos a serem processados, teremos $log(T/n)$ níveis. Por exemplo, se tivéssemos n postagens, apenas a lista auxiliar seria necessária. Com isso, a inserção de uma postagem ao índice terá complexidade $\Theta(log(T/n))$, pois passará no máximo por todos os níveis. A indexação de todas as postagens, por sua vez, terá complexidade $\Theta(Tlog(T/n))$.

O problema desse método é que uma busca terá que percorrer todos os níveis mesclando os resultados e não só as duas listas que tínhamos anteriormente. Pode ser preferível, portanto, reconstruir o índice de tempos em tempos, dependendo da aplicação.

2.5 Pontuação, ponderação do termo e modelo do espaço vetorial

2.5.1 Atribuir pontuação a um documento

Para ranquear os resultados de uma busca é necessário algum sistema de pontuação. Um documento que menciona um termo da consulta mais vezes é mais interessante, logo, deve receber uma pontuação maior. Essa pontuação de um termo t em um documento d será chamado de peso de t em d. Pensando em consultas de texto livre, que é uma forma mais comum na web, seria fácil computar a pontuação de um documento, sendo esta soma dos pesos dos termos da consulta presentes no documento. Segundo Manning, Raghavan e Schütze (2008), a forma mais natural de atribuir esse peso é pela frequência do termo, denotada $tf_{t,d}$, mas pode-se usar qualquer função de ponderação. Essa forma é conhecida como modelo saco-de-palavras (do inglês, *bag of words model*) que leva em consideração apenas o número de ocorrências dos termos, mas não a ordem destes. Mesmo que a ordem das palavras divirja entre dois documentos, se as frequências dos termos forem similares, esses são considerados similares.

Alguns termos podem ser muito frequentes na coleção, logo, não terão muita influência no ranqueamento dos documentos, sendo necessária algum tipo de atenuação. Utilizando a frequência do termo na coleção, um termo muito presente em poucos documentos e um termo pouco presente em muitos documentos teriam a mesma frequência, então, a atenuação não surtiria efeito. Portanto, é comum usar a frequência de documentos (df_t), ou seja, o número de documentos da coleção em que o termo está presente. Assim, definimos frequência inversa do documento (idf):

$$idf_t = log\frac{N}{df_t}$$

Um termo que está em todos os documentos terá $idf = 0$. Quanto mais raro o termo na coleção, maior será o seu idf. Multiplicando $tf_{t,d}$ por idf_t temos um balanceamento entre a presença de um termo em um documento e a sua raridade na coleção. Um documento que diz muito sobre um assunto que aparece pouco na coleção deve ser mais relevante para este assunto. Definimos então o balanceamento tf-idf:

$$tf\text{-}idf_{t,d} = tf_{t,d} \cdot idf_t$$

Trabalhando com o documento como um vetor de pesos para os termos do dicionário, temos a

pontuação de um documento d em relação à uma consulta q:

$$score(q,d) = \sum_{t \in q} tf\text{-}idf_{t,d}$$

Esta representação de documentos como vetores em um espaço vetorial é conhecida como modelo vetorial (do inglês, *vector space model*) e pode ser usado em outras operações de recuperação de informação. Além disso, outras funções de ponderação podem ser usadas no lugar de tf-idf. Para cada documento d temos, então, um vetor $\overline{V}(d)$ com componentes sendo as ponderações. Assim, o espaço vetorial terá um eixo para cada termo.

Dois documentos serão similares se tiverem vetores com componentes proporcionalmente parecidas, mesmo que as componentes de um documento sejam maiores do que a do outro. Para computar esta similaridade, é comum o uso do cosseno entre os dois vetores:

$$sim(d_1,d_2) = \frac{\overline{V}(d_1) \cdot \overline{V}(d_2)}{|\overline{V}(d_1)| \cdot |\overline{V}(d_2)|}$$

Temos a divisão do produto vetorial entre os vetores sobre o produto de seus comprimentos euclidianos. Esta divisão é necessária para normalizar os resultados. Se os vetores já estiverem normalizados, apenas o produto vetorial é necessário.

A coleção será representada pelo conjunto destes vetores: uma matriz MxN, sendo M o número de termos do dicionário e N o número de documentos. Da mesma forma que um documento, uma consulta pode ser representada por um vetor. Sendo assim, a pontuação de um documento em relação a uma consulta será:

$$score(q,d) = \frac{\overline{V}(q) \cdot \overline{V}(d)}{|\overline{V}(q)| \cdot |\overline{V}(d)|}$$

Além do uso do cosseno para o cálculo da similaridade, outras funções podem ser usadas. Busca-se, hoje, determinar de maneira mais formal a influência destas funções em diferentes domínios.

2.5.2 Índices paramétricos e de zonas

Buscas poderão depender de outras informações que estão além do corpo de um documento. Como alternativa, Moura, Pereira e Campos (2002) propõem o uso de metadados como uma solução adequada para promover uma recuperação de recursos na Web mais eficiente e precisa, já que permite um melhor agrupamento de recursos digitais. Por exemplo, poderíamos querer resgatar documentos que foram criados antes de uma data ou por um autor específico. Guardando estas informações (data de publicação e autor, no exemplo) como metadados dos documentos, podemos recuperá-los com precisão. Manning, Raghavan e Schütze (2008) definem estas representações das informações nos metados como campos. Para a indexação destes campos, os autores dão como possibilidade o uso de índices paramétricos (do inglês, *parametric indexes*). Nestes, para cada campo é construída uma lista invertida com suas postagens. Assim, fazemos buscas separadas e mesclamos os resultados.

Os campos de metadados podem ter em um domínio de valores ordenáveis, como datas, por exemplo. Assim, o dicionário deste campo pode ser implementado em uma estrutura que facilite a navegação por ela, como uma árvore binária.

Além dos campos, uma busca pode ser feita em partes específicas de um documento, como o título ou um resumo. Para tal, utiliza-se zonas, similares aos campos, mas que assumem textos de tamanho arbitrário. Para cada zona podemos construir um índice, cujo dicionário é constituído dos termos presentes nesta zona. Outra alternativa é codificar a zona em que um termo aparece e fazermos um índice único, o que reduz o espaço ocupado pelo dicionário.

Até aqui utilizamos os índices paramétricos para fazermos buscas em que um termo está ou não presente em uma zona. Podemos, então, ponderar as zonas e atribuir uma pontuação para cada documento. Por exemplo, talvez seja uma informação mais relevante para o usuário se um termo

aparece no corpo de um documento, enquanto o título pode não ser importante. Se tivermos n zonas, e atribuirmos um peso p_i para cada zona, podemos normatizar a soma dos pesos para que:

$$\sum_{i=1}^{n} p_i = 1$$

Dado uma query q e um documento d, podemos atribuir uma pontuação s_i para cada zona do documento. A pontuação total do documento é, portanto, a combinação linear dos pesos e as pontuações das zonas:

$$score(d, q) = \sum_{i=1}^{n} p_i s_i \tag{2.1}$$

Um problema surge, naturalmente: como determinar os pesos p_i? Eles podem ser determinados pelo próprio usuário ou "aprendidos" através de exemplos escolhidos, ou seja, o aprendizado de máquina. Os autores definem os exemplos como tuplas que contém uma query q, um documento d e um julgamento da relevância, que pode ser tão simples quanto "é relevante" e "não é relevante", mas pode ter outras nuâncias. Através destes exemplos, a máquina gera pesos p_i que tenham resultados que se aproximem dos julgamentos de relevância que foram fornecidos. Isto pode ser escrito como um problema de otimização.

No aplicativo *LookingFor* desenvolvido neste Trabalho de Conclusão de Curso, exploramos o primeiro caso: o usuário determina os pesos para cada consulta. No entanto, nenhum parâmetro textual é fornecido. O sistema utiliza parâmetros preestabelecidos para gerar as pontuações. Ou seja, o usuário influencia os pesos p_i mas não as pontuações s_i.

Capítulo 3

Desenvolvimento

Como apresentado na Seção de Introdução, além de existir poucas formas de parametrização de buscas na Web, não há a possibilidade de o usuário ponderar esses parâmetros. Como alternativa, foi desenvolvido o aplicativo *LookingFor*. Esse sistema permite que o usuário ordene os parâmetros da busca de acordo com a sua necessidade. O aplicativo transforma essa ordenação em pesos, que são utilizados na busca parametrizada. Como os resultados não são apenas filtrados, utilizamos os pesos para pontuar os documentos, destacando resultados que podem ser mais relevantes para o usuário. A estrutura desse aplicativo pode ser vista na figura 3.1.

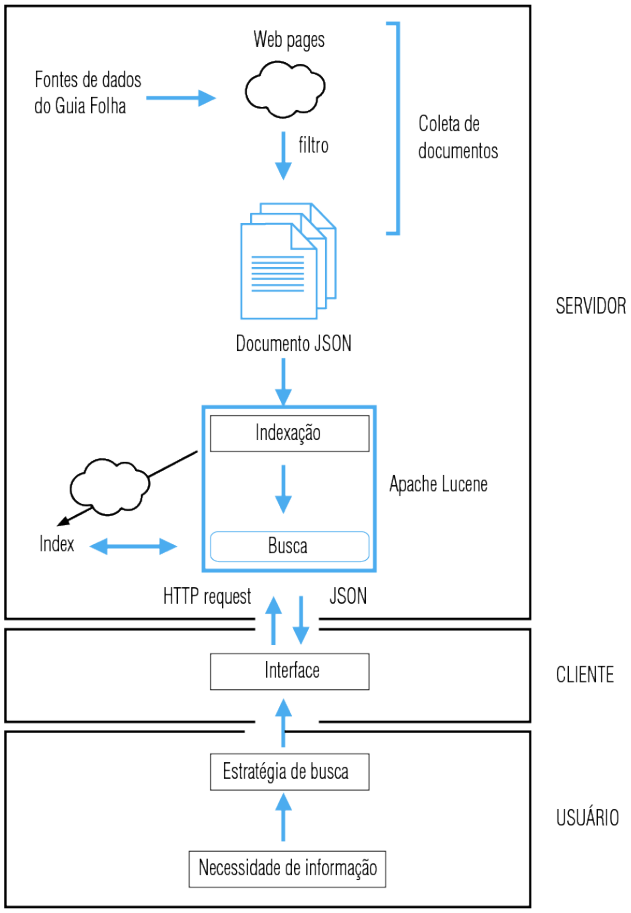

Figura 3.1: *Diagrama do funcionamento do sistema.*

Capturamos documentos HTML da Web e extraímos dados relevantes para as buscas. Para armazenar essas informações, são criados arquivos de metadados. Com a coleção construída, pudemos indexá-la e realizar buscas com auxílio do software open source Apache Lucene [Apaa]. Por fim, os resultados das buscas são apresentados para o usuário através de uma interface.

3.1 Coleta de documentos e metadados

Para coletarmos documentos, utilizamos um *script Shell* (aplicação criada para interagir com o sistema operacional). Esse *script* acessa e baixa o conteúdo de uma lista de links preestabelecida, gerando documentos locais. Alternativamente, um *Web Crawler* poderia ter sido utilizado nesta etapa. Segundo Manning, Raghavan e Schütze (2008), *Web Crawling* é o processo de coletar páginas da Web, para criar um índice de uma ferramenta de busca. Também é conhecido como indexador automático. Duas alternativas são os softwares livres HTTrack [HTT] e Apache Nutch [Apab].

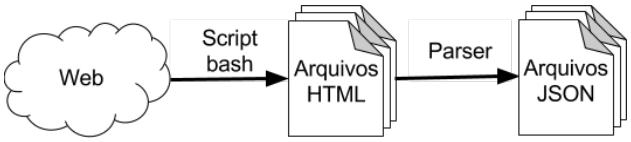

Figura 3.2: *Diagrama da coleta de documentos da Web.*

A lista de links foi extraída do site Guia Folha [Gui]. Escolhemos essa fonte, pois suas páginas seguem um padrão de estrutura, contendo dados em *tags* bem definidas do código HTML.

Após a coleção ter sido criada, executamos uma etapa de limpeza nos documentos HTML. Retiramos todas as *tags* do documento, pois apenas as informações textuais desses documentos têm valor para o usuário. Paralelamente, extraímos dados presentes nos documentos para gerar seus metadados. Para isso utilizamos expressões regulares [Exp] em partes específicas dos documentos, buscando por conjuntos de caracteres e palavras específicos. Por exemplo, após a expressão "Telefone:", uma sequência de dígitos representa o telefone do estabelecimento.

Armazenamos esses metadados no formato JSON (*JavaScript Object Notation*) [JSO]. Utilizamos esse formato por estarmos mais familiarizados. Outras notações poderiam ser utilizadas como o XML (*Extensible Markup Language*) [XML].

3.2 Indexação

Como dito anteriormente, utilizamos o software Apache Lucene [Apaa] para indexar a coleção. O Lucene é uma biblioteca para buscas textuais, sendo amplamente utilizado em sistemas de RI. Possui operações de indexação e busca para coleções de documentos de texto.

A classe *IndexWriter* [Ind] é a responsável por adicionar os documentos ao índice. O índice pode ser armazenado em disco ou em memória, utilizando as classes *FSDirectory* e *RAMDirectory*, respectivamente. Nesta implementação, optamos por guardar o índice no disco, criando novos processos para cada busca.

Para representar os documentos durante a indexação é utilizada a classe *Document*, que armazena suas zonas em objetos da classe *Field*. Dessa forma, é fácil recuperar esses campos individualmente, separando os índices para cada zona que serão a base para a busca parametrizada. Adicionalmente, esses campos podem ser acessados na etapa de busca para atribuir pontuações a dados não-textuais, em que não se aplicam modelos de similaridade como o tf-idf.

Cada documento é processado pela classe *DocConsumer* e armazenado em memória. Essas operações podem ser realizadas por várias threads, cada uma mantendo um índice parcial. Através de um sistema de eventos, a classe *MergePolicy* é chamada para mesclar esses índices parciais ao principal, mantendo sua consistência. A instância da classe *MergePolicy* é única e serve de ponte entre as diferentes threads e o disco, concentrando os acessos em uma só operação. Com isso, não se desperdiça espaço nos blocos transferidos ou tempo de processamento, como visto na seção 2.4.

Criamos uma classe adicional *IndexFiles* para encapsular a indexação de uma coleção. Nela, é aberto o diretório em que se encontra a coleção, iterando os documentos para adicioná-los ao índice, utilizando a classe *IndexWriter*. Para cada documento, utilizamos a classe *Field* para representar seu corpo e seus metadados. Três desses campos são: a faixa de preço do restaurante, as suas coordenadas geográficas e a qualidade do restaurante (uma nota atribuída previamente). Esses valores serão utilizados na etapa de busca.

3.3 Busca parametrizada

Buscas parametrizadas nos permitem um refinamento da busca, analisando mais aspectos de um documento. Como visto, podemos fazer buscas em várias zonas ou campos, procurando informações diferentes. A análise de cada zona nos fornece pontuações separadas que compõem uma nota final dada ao documento. Para isso, são utilizados índices paramétricos. Como dito na seção 2.5, um índice paramétrico é composto por outros índices, um para cada zona.

Para zonas ou campos com informações puramente textuais, modelos de similaridade como o *tf-idf* são utilizados. Obviamente, para campos com outros domínios, esses modelos não se aplicam. Por exemplo, se buscarmos pelo termo "1992" e um documento tiver o termo "1993", esse documento não terá uma pontuação, pois a frequência do termo "1992" é zero. Em outras palavras, os termos "1992" e "1993" são considerados diferentes.

Se analisarmos esses termos como números inteiros, entretanto, podemos dar alguma pontuação a esse documento. Ocorrências do termo "1992" podem receber a pontuação 1, por estar o mais próximo possível do termo procurado. O documento com o termo "1993" receberá uma nota menor, por exemplo, 0,9. Ou seja, podemos utilizar uma função com domínio nos inteiros e imagem no intervalo [0, 1].

Na aplicação *LookingFor*, o usuário pode fazer uma consulta em linguagem natural e configurar três parâmetros: distância, preço e qualidade. Essa configuração é feita sem utilizar palavras ou faixas de valores disponíveis em filtros. Como já foi dito, o usuário fornece ao sistema uma ordem de prioridade para esses parâmetros, tentando comunicar a sua necessidade, como pode ser visto na figura 3.3.

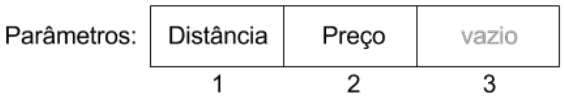

Figura 3.3: *Lista de prioridades para os parâmetros.*

Essa ordem de prioridade será traduzida em pesos, utilizados para ponderar as pontuações das zonas e campos, como na equação 2.1:

$$score(d, q) = \sum_{i=1}^{n} p_i s_i$$

Para isso, definimos o conjunto de parâmetros P, com os parâmetros distância, preço e qualidade:

$$P = \{dist,\ preço,\ qualid\}$$

Através da interface de usuário, é gerado um vetor v sem repetição com até $|P|$ elementos, como na figura 3.3. Com isso, definimos o peso do corpo do documento:

$$p_{corpo} = 1 + |v|$$

E os pesos dos parâmetros:

$$p_{v[j]} = p_{corpo} - j$$

Utilizando o exemplo da figura 3.3, os valores dos pesos seriam: $p_{corpo} = 3$, $p_{dist} = 2$, $p_{preço} = 1$ e $p_{qualid} = 0$. Podemos também normalizar os pesos, de forma que:

$$\sum_i p_i = 1$$

Como utilizamos o modelo saco-de-palavras, e considerando a consulta como um pequeno documento, a pontuação representa a similaridade entre a consulta e o documento que está sendo avaliado. Os parâmetros, no entanto, não são definidos pelo usuário, mas por valores hipotéticos atribuídos pelo sistema. Os usuários apenas definem o texto da consulta e os pesos desses parâmetros na busca.

Encaramos a consulta como um documento ideal d', o que para o nosso sistema significa um estabelecimento muito próximo do local do usuário, muito barato e com a melhor qualidade possível. Vamos analisar cada um desses parâmetros individualmente.

Para o critério qualidade, utilizamos uma enumeração para criar uma relação de ordem entre os elementos. Em nosso estudo de caso, os elementos são: "ruim", "regular", "bom", "muito bom" e "ótimo". Portanto, d' tem o maior valor de qualidade dentre esses elementos: "ótimo". Quanto menor for o valor para um documento, mais distante estará do ideal e menor será a sua pontuação.

$$s_{qualid,\ d} = \frac{1}{qualidade(d')} qualidade(d)$$

O segundo critério que utilizamos é o preço médio dos estabelecimentos. No nosso estudo de caso, uma tupla de valores monetários estava associada a cada documento. Essa tupla define o intervalo de preços do restaurante. Para simplificar a análise, utilizamos a média desses dois valores para o cálculo da similaridade. O valor de preço médio para o d' é R$ 0,00, ou seja, é um estabelecimento gratuito. Logo, os estabelecimentos com preços mais baixos terão pontuações melhores.

$$s_{preço,\ d} = 1 - \frac{1}{max(preço(d_1), ..., preço(d_n))} preço(d)$$

Por fim, para o critério distância, utilizamos coordenadas geográficas. Cada estabelecimento possui um endereço, que foi traduzido em coordenadas na etapa de indexação. Para isso, utilizamos a API do GoogleMaps. Para essas coordenadas, a distância entre dois pontos é calculada sobre o globo terrestre. O documento d' possui as mesmas coordenadas geográficas do usuário, ou seja, ele não precisará se locomover. Os estabelecimentos mais próximos do usuário terão pontuações maiores.

$$s_{dist,\ d} = 1 - \frac{1}{max(distância(d_1, d'), ..., distância(d_n, d'))} distância(d, d')$$

Como fica claro, nenhuma dessas pontuações é atenuada. Deixamos isso para os trabalhos futuros. Por exemplo, usar uma função logarítmica ao invés de uma equação de primeiro grau.

Avaliando cada documento sobre esses critérios, podemos finalmente determinar a pontuação final:

$$score(d,\ q) = p_{corpo} \left(\sum_{t \in q} tf - idf_{t,d} \right) + \sum_{i \in P} p_i s_i$$

Encapsulamos o processo de busca e ranqueamento na classe *SearchFiles*, criada sobre o arcabouço Lucene. Essa classe recebe os parâmetros escolhidos pelo usuário, utiliza a busca do Lucene e calcula a similaridade para cada parâmetro. Por fim, ordena os resultados por esse ranqueamento antes de devolvê-los ao usuário.

3.4 Interface de usuário e georreferenciamento

O aplicativo *LookingFor* foi desenvolvido como um sistema Web. A interface de usuário pode ser vista na figura 3.5. Ela consiste de três elementos: o formulário de busca, o mapa de georreferenciamento e a lista de resultados. Essas partes podem ser vistas na figura 3.5.

A estrutura da consulta é representada no formulário de busca. Ela possui um campo de texto e

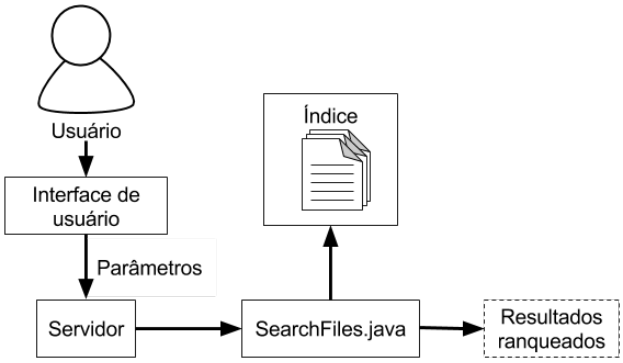

Figura 3.4: *Diagrama do processamento de uma busca.*

um botão para cada parâmetro. Ao clicar em um dos botões, um número aparece ao seu lado. Esse número corresponde a prioridade desse parâmetro na busca. Quanto mais baixo o número, maior é o seu peso, como descrito na sessão 3.3. Os parâmetros que não forem escolhidos terão peso zero. Conforme o usuário seleciona e desfaz essas seleções os pesos são ajustados automaticamente por uma função javascript, presente na própria interface.

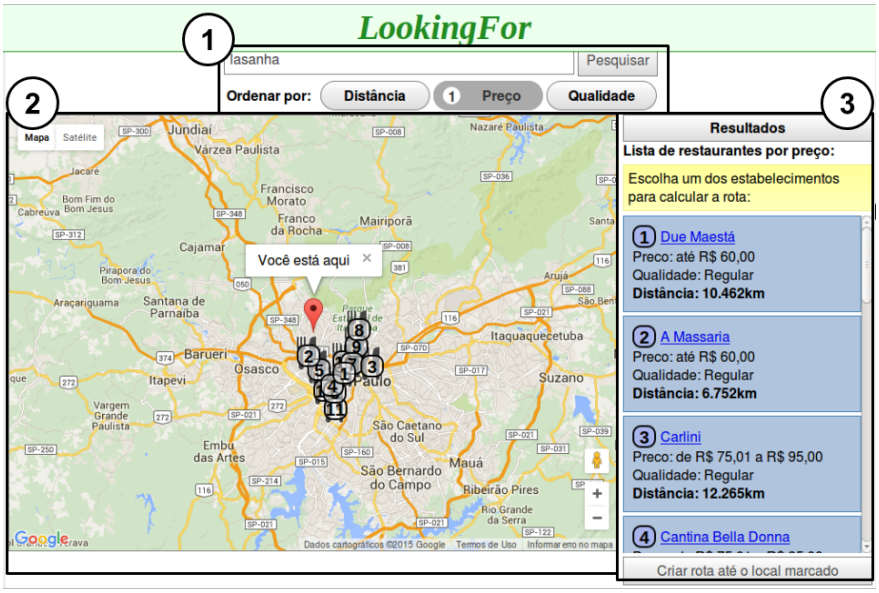

Figura 3.5: *Interface do usuário*

Quando o usuário envia o formulário, uma requisição HTTP é feita para o servidor utilizando o padrão AJAX (*Asynchronous Javascript and XML*). A busca é feita no índice e os resultados retornados no formato JSON. Os estabelecimentos são extraídos como objetos javascript e apresentados na lista de resultados à direita na tela. Nessa lista, são apresentadas algumas informações sobre cada estabelecimento: o endereço, a qualidade e a faixa de preço. Dessa forma, o usuário pode entender mais claramente a influência dos parâmetros na busca.

Os resultados também são representados no mapa à esquerda da tela, através do georreferenciamento de seus endereços físicos. Segundo Meireles, Almeida e Silva (2009) "Georreferenciar é atribuir coordenadas a um ponto de um mapa, vinculando-o a um sistema de coordenadas real." O georreferenciamento é utilizado por alguns sistemas para ajudar na compreensão das distâncias entre resultados e sua localização em relação a algum referencial. Essa forma de apresentar os resultados é mais visual, por transformar endereços em pontos em um sistema de coordenadas.

Capítulo 4

Aplicação

Como dito no capítulo 3, utilizamos o site GuiaFolha como nosso estudo de caso. Escolhemos esse sistema por ter dados de fácil extração. No entanto, a base de dados da aplicação *LookingFor* [Rep] pode ser estendida, recebendo outras fontes de dados. Bastaria adaptar os dados de cada fonte ao nosso modelo, que já foi apresentado.

No site GuiaFolha, estão catalogados cerca de 950 restaurantes da cidade de São Paulo. Muitos desses sites não possuem endereços eletrônicos associados, apenas uma breve descrição. Outras informações são apresentadas de forma estruturada, como tipo de cozinha, endereço, telefone, preço médio e o horário de funcionamento. O formulário de busca desse site pode ser visto na figura 4.1. O usuário consulta o sistema utilizando um campo de texto e algumas caixas de seleção.

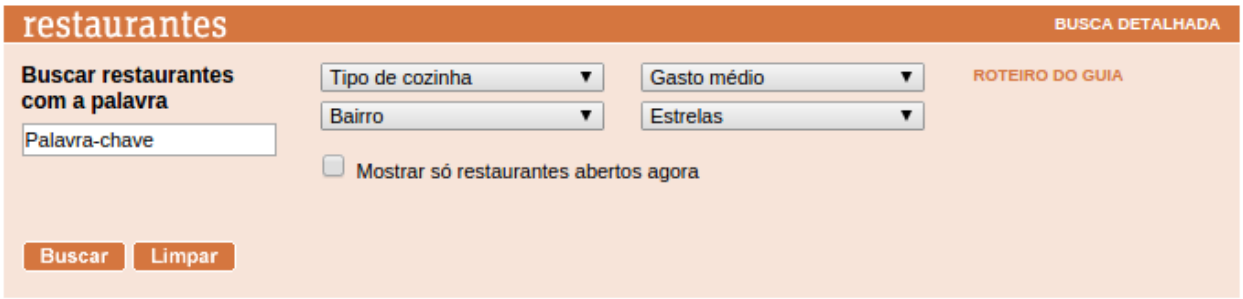

Figura 4.1: *Formulário de restaurantes do site GuiaFolha.*

Para simplificar o nosso estudo, unimos o tipo de cozinha e a descrição para formar o corpo do documento. Os outros dados foram armazenados como metadados.

No aplicativo *LookingFor*, também fornecemos uma caixa de texto para o usuário expressar a sua necessidade. No entanto, trocamos as caixas de seleção "gasto médio", "bairro" e "estrelas" pelos botões "preço", "distância" e "qualidade", respectivamente. Com isso, simplificamos a interface de busca, mantendo certa liberdade. Esperamos que esse novo formato torne a formulação das consultas mais rápida e menos frustrante para o usuário.

Apresentamos esse aplicativo para um grupo de 19 usuários, guardando alguns dados sobre os seus acessos. Além disso, pedimos que eles respondessem um breve questionário, relatando a sua experiência.

4.1 Resultados

Utilizamos um sistema de gerenciamento de banco de dados para guardar as informações dos acessos dos usuários. O software MySQL foi escolhido por termos maior familiaridade. Duas tabelas foram criadas: buscas e escolhas.

A tabela "buscas" foi usada para registrar as consultas feitas no sistema. Nela estão quatro colunas principais: *text*, *distance*, *price*, *grade*. A coluna *text* representa os termos da consulta,

como digitados pelo usuário. As outras três colunas guardam as prioridades dos parâmetros da busca. Essas colunas terão valor nulo se não forem escolhidos pelo usuário. Caso contrário, terão como valor um número natural menor ou igual ao número total de parâmetros.

A outra tabela, "escolhas", guarda a posição do resultado escolhido por um usuário. Partimos da hipótese que, quando um usuário clica em algum resultado, então este pode ser interessante. Não necessariamente esse interesse se traduz em relevância, mas pode ser um indicador. Algumas evidências presentes nesses resultados podem contribuir para entendermos o que o usuário necessita.

Para registrar essa escolha, apenas uma coluna é utilizada, chamada "posição". Quanto mais próximo o resultado está do topo da lista de resultados, melhor é a sua colocação. Ou seja, maior foi a pontuação dada pelo sistema de RI. Se muitos usuários escolhem resultados longe do topo, isso pode indicar falhas no nosso sistema de ranqueamento.

4.1.1 Consultas

Nesse período de testes, 253 consultas foram realizadas. Em média 11,8 consultas por usuário. Muitos termos foram repetidos nas consultas, de forma que apenas 87 distintas foram feitas, desconsiderando os parâmetros escolhidos. Dessas, 40 (46%) não retornaram nenhuma resposta.

Duas hipóteses podem ser feitas sobre esse fato. Primeiro, por termos usado a descrição breve dos estabelecimentos do site GuiaFolha, o dicionário do sistema foi construído com poucos termos. Segundo, por termos usado o modelo saco-de-palavras sem nenhum processamento linguístico, muitos termos próximos não foram associados durante a busca. Por exemplo, a consulta "japonesa" retorna 27 resultados, mas "japonês" não retorna nenhum. É importante para um sistema de RI compreender a linguagem utilizada pelos seus usuários, para que possa satisfazer a necessidade de informação sem um maior esforço por parte do usuário.

Considerando os parâmetros utilizados nas consultas, podemos entender que aspectos dos estabelecimentos interessam mais os usuários. A frequência de cada parâmetro pode ser vista na figura 4.2. Cada barra é segmentada pela colocação do parâmetro na ordem de prioridade.

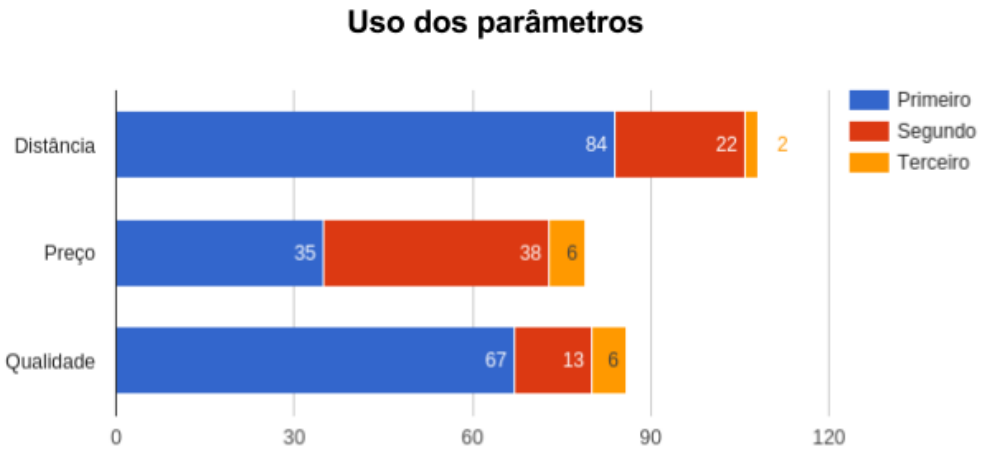

Figura 4.2: *Gráfico da frequência dos parâmetros nas consultas.*

O parâmetro distância foi o mais utilizado, estando presente em 108 consultas. Uma diferença de 20,4% para o segundo colocado. No entanto, isso não quer dizer que os usuários tenham uma maior necessidade por estabelecimentos próximos ao seu local. A interface de usuário do aplicativo apresenta um mapa a todo momento, mesmo antes dos resultados da busca serem apresentados. Isso pode estimular o usuário a fazer consultas com conceitos geográficos, como a distância.

O preço foi o menos comum dos parâmetros, mas foi o mais utilizado como segundo na ordem de prioridade. Essa pode ser uma necessidade menor do usuário, mas ainda é um parâmetro relevante.

Todos os três parâmetros foram usados poucas vezes no final da ordem de prioridade. Podemos ver isso também na figura 4.3, a qual mostra o número de parâmetros por consulta. 94,5% delas

utilizaram até dois parâmetros. Considerando a distribuição das frequências na figura anterior, entendemos que os três parâmetros são relevantes para os usuários, mas não todos ao mesmo tempo.

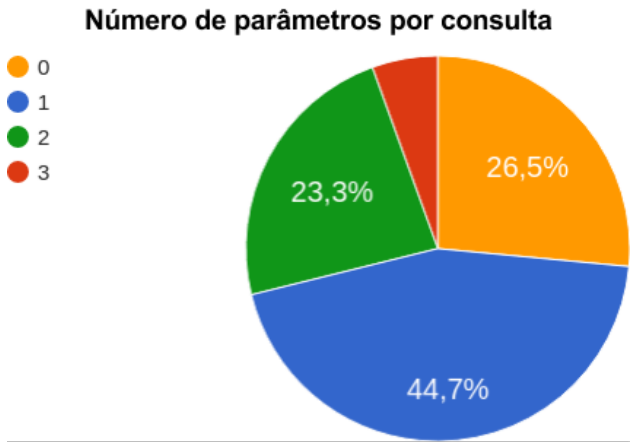

Figura 4.3: *Gráfico do número de parâmetros por consulta.*

Já que os usuários foram introduzidos ao aplicativo para realizar este experimento, provavelmente tentaram entender o sistema de busca e o significado de cada parâmetro no primeiro momento. Para isso, realizaram consultas semelhantes, comparando seus resultados. Isso pode explicar a grande proporção de consultas com apenas um deles selecionado. Para resultados mais conclusivos, seria necessário expôr os usuários ao aplicativo por um tempo maior, deixando-os se familiarizar com as funcionalidades.

4.1.2 Estabelecimentos escolhidos

Rastreando os cliques feitos na lista de estabelecimentos do aplicativo, foi possível guardar a posição do elemento escolhido. Contudo, no período de testes, apenas 39 desses cliques foram feitos. Isso representa 24,5% das 163 buscas com algum resultado.

Como já foi dito, os usuários não acessaram o *LookingFor* por conta própria, mas foram convidados a fazê-lo. É possível que para esse grupo não houvesse uma real necessidade das informações que o aplicativo pode prover, estando mais preocupados em explorar a interface de usuário, gerando poucos cliques na lista de resultados.

Como pode ser visto na figura 4.4, 17 dos 39 estabelecimentos escolhidos estavam no topo da lista, e existe uma preferência pelos primeiros resultados.

O aplicativo retornou em média 18,4 resultados por busca, considerando apenas as que retornaram algo. Se esse número fosse menor, poderíamos argumentar que as posições mais baixas tiveram uma frequência maior por serem as únicas possíveis na maioria dos casos.

Figura 4.4: *Frequência de cliques por posição da resposta.*

A posição média dos estabelecimentos escolhidos é 4,5. Em um monitor 1680x1050, a lista de resultados mostra entre 4 e 5 elementos, sem rolagem. Portanto, os usuários podem ter clicado nesses primeiros por estarem visíveis. Além disso, uma posição é sempre menor ou igual ao tamanho da lista, logo, os números menores estarão em mais listas. A primeira posição, por exemplo, está em todas.

Como foi dito antes, com um período de testes mais longos conseguiríamos avaliar melhor a reação dos usuários ao aplicativo.

4.2 Feedback dos usuários

Foi aplicado um questionário estruturado (apêndice A) com 4 perguntas, as quais nenhuma era obrigatória. Como já foi dito, 19 pessoas foram convidadas a participar. Elas podem ser divididas em dois conjuntos de acordo com a idade. O primeiro com 15 pessoas entre 20 e 30 anos. O segundo com 4 pessoas de 54 a 61 anos. No entanto, essa diferença não se mostrou perceptível em suas respostas.

Na primeira questão, perguntamos ao usuário se alguma de suas consultas não retornou resultados. Daqueles que responderam, 13 responderam afirmativamente. Mesmo para dois termos próximos como "massas" e "massa", um obtinha resultados e o outro não. Isso reafirma o que foi dito na seção anterior: com o modelo saco-de-palavras, muitas consultas têm um *recall* baixo, pois mesmo quando sabemos que há documentos relevantes no sistema, nenhum é retornado.

Na segunda questão, perguntamos o quanto os usuários acharam que os parâmetros foram relevantes em suas consultas. Podemos ver os resultados na figura 4.5.

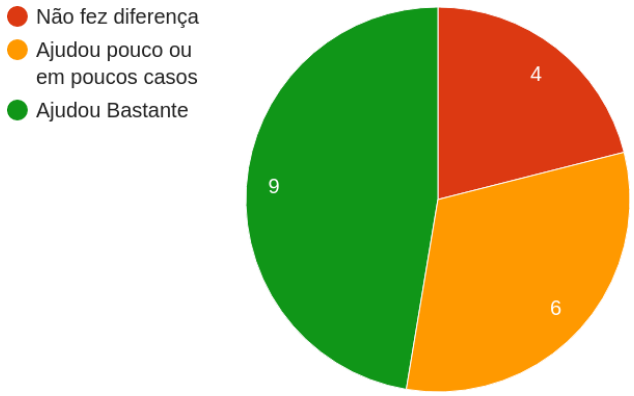

Figura 4.5: *Respostas para a segunda pergunta do questionário.*

Nenhum usuário respondeu que os parâmetros dificultaram a busca. Houve uma divisão entre aqueles consideraram o sistema útil e aqueles que não identificaram muitas vantagens em seu uso. Novamente, dado que o sistema tem poucos documentos e as consultas têm um baixo *recall*, talvez a diferença entre as listas de resultados não fosse tão perceptível.

Na terceira pergunta, queríamos saber se os usuários colocariam ou retirariam algum parâmetro. 9 responderam que não fariam modificações. Outros 7 deram algumas sugestões. Muitas envolviam acessibilidade e transporte, como "horário de funcionamento", "número de vagas" e "estacionamento/transporte público próximos". Apenas 1 usuário respondeu que retiraria o parâmetro "qualidade", por achá-lo pouco confiável em um aplicativo tão novo.

A quarta e última pergunta pedia a opinião dos usuários quanto a lista de resultados e se nela faltavam informações. As respostas eram semelhantes a da questão anterior, com alguns itens novos como: "opção de reserva", "aceita vale refeição", "música ao vivo".

Para os trabalhos futuros, essas duas últimas questões são relevantes, pois mostram necessidades dos usuários que não são atendidas. Algumas dessas sugestões já são implementadas em outros sistemas, como o Kekanto ou o GoogleMaps.

Por fim, deixamos um campo para comentários, caso as pessoas quisessem comunicar algo que não estava nas perguntas. Recebemos algumas críticas quanto ao layout do aplicativo e alguns erros que aconteceram. Por exemplo, a aplicação não funcionou corretamente em dispositivos móveis.

Um dos comentários criticava o ranqueamento dos resultados, pois eles não eram totalmente reordenados. Portanto, o sistema de ponderamento pode não ter ficado claro para alguns usuários. Algumas mudanças de layout poderiam ajudar nesse sentido.

Capítulo 5

Conclusões

Recuperação de Informação é uma área da Ciência da Computação de extrema relevância, pois métodos de recuperação sobre dados não-estruturados (como texto, imagens, entre outros) contidos em documentos digitais (e.g. páginas web) são pré-requisitos para o tratamento de grandes volumes de dados. RI está presente nas ferramentas do dia-a-dia das pessoas – e-mails, sites de busca, comércios eletrônicos, entre outros. Um dos desafios da área é apoiar o processo de descoberta de conhecimento num contexto de crescimento exacerbado de dados na Internet.

Dentre os conceitos de RI, três se destacam: indexação, processamento linguístico e ranqueamento. A indexação é necessária para construir o índice invertido antes de qualquer busca ser requisitada pelo usuário. Essa estrutura de dados tem como principal objetivo armazenar listas de postagens (ordenadas pelo id do documento) em disco e associar cada lista a um termo específico do dicionário, e este por sua vez normalmente é mantido em memória principal. Os termos do dicionário são normalmente ordenados lexicograficamente.

Já o processamento linguístico é usado para auxiliar na construção do dicionário, ao gerar termos que representam classes de tokens de mesma base léxica. Tal método é conhecido como normalização e dois tipos de algoritmos podem ser utilizados como de exemplo: stemização e lematização.

Por último, e não menos importante, está o ranqueamento, cuja base é o cálculo do tf-idf, sendo adaptado conforme cada aplicação. Ele é útil na ponderação dos documentos a fim de ordená-los por relevância em função da consulta.

Um outro desafio de RI é definir o conceito de relevância, dado que o usuário final é quem aprova as respostas retornadas e não o projetista do sistema de RI. O *LookingFor*, aplicativo desenvolvido no nosso Trabalho de Conclusão de Curso, permite combinar o campo de texto com alguns filtros adicionais que incluem parâmetros de interesse do usuário. O objetivo foi alcançado, já que a busca do usuário tornou-se mais cômoda e otimizada. No entanto, alguns ajustes serão necessários, mas eles surgirão à medida que estatísticas de controle forem geradas pelas ações dos usuários.

Como trabalhos futuros, o aplicativo poderá ser estendido para outros estabelecimentos como também para outras categorias de coleções, como livros de uma biblioteca digital. Além disso, poderão ser inclusos parâmetros de diversas naturezas - tais como idade, gênero, índice de violência, etc – de acordo com uma análise mais apurada da real necessidade dos usuários.

Apêndice A

Formulário

Looking For - Experiência do usuário Todas as perguntas são opcionais. Ajude-nos como puder! Lembrando que resultados negativos e críticas nos ajudarão também. Gênero (Feminino/Masculino/Outro) Idade (Livre)

1. Houve alguma pesquisa que você fez em que não obteve resultados? Que termos você utilizou?
2. Você acha que os parâmetros lhe ajudaram a encontrar resultados interessantes? (Atrapalhou/Não fez diferença/Ajudou pouco ou só em alguns casos/Ajudou bastante)
3. Você gostaria de acrescentar ou retirar algum parâmetro da busca? (Parâmetros existentes: preço, distância e qualidade.)
4. Nos resultados da busca, você sentiu falta de alguma informação referente aos estabelecimentos?

Comentários ou sugestões? Por exemplo: sobre o layout; algo que você achou confuso ou difícil de entender.

Referências Bibliográficas

[Apaa] Apache Lucene. https://web.archive.org/web/20151102014915/http://lucene.apache.org/core/. Accessed: 2015-11-2. 14, 15

[Apab] Apache Nutch. http://nutch.apache.org/. Accessed: 2015-11-27. 15

[EJ08] Nuno Filipe Escudeiro e Alípio Mário Jorge. Satisfying Information Needs on the Web: a Survey of Web Information Retrieval. *5. Tékhne - Revista de Estudos Politícos*, páginas 337 – 369, 06 2008.

[Exp] Artigo sobre expressões regulares. https://pt.wikipedia.org/wiki/Express%C3%A3o_regular. Accessed: 2015-11-27. 15

[Gooa] Google. https://web.archive.org/web/20151122162241/http://www.google.com.br/. Accessed: 2015-11-22. 2

[Goob] Google Maps. https://web.archive.org/web/20150429170137/https://www.google.com.br/maps. Accessed: 2015-11-22. 2

[Gui] Guia Folha. https://web.archive.org/web/20150908233920/http://guia.folha.uol.com.br/restaurantes/. Accessed: 2015-11-22. 2, 15

[HTT] HTTrack. http://www.httrack.com/. Accessed: 2015-11-27. 15

[Ind] Documentação da classe IndexWriter. https://web.archive.org/web/20150515144032/http://lucene.apache.org/core/4_0_0/core/org/apache/lucene/index/IndexWriter.html. Accessed: 2015-05-15. 15

[JSO] Artigo sobre JSON. https://web.archive.org/web/20150911133904/https://pt.wikipedia.org/wiki/JSON. Accessed: 2015-11-27. 15

[Kek] Kekanto. https://web.archive.org/web/20151009101731/https://kekanto.com.br/. Accessed: 2015-11-22. 2

[MAS09] Magali Rezende Gouvêa Meireles, Paulo Eduardo Maciel de Almeida e Ana Carolina Milagres Resende Silva. 2. Recuperação de informação no ambiente acadêmico: 2. georreferenciamento dos dados dos estudantes do Instituto de Educação Continuada da PUC Minas. *Perspectivas em Ciência da Informação*, 14:61 – 74, 12 2009.

[MPC02] Ana Maria de Carvalho Moura, Genelice da Costa Pereira e María Luiza Machado Campos. A metadata approach to manage and organize electronic documents and collections on the web. *Journal of the Brazilian Computer Society*, 8:16 – 31, 07 2002.

[MRS08] Christopher D. Manning, Prabhakar Raghavan e Hinrich Schütze. *Introduction to Information Retrieval*. Cambridge University Press, New York, NY, USA, 2008.

[RC11] Bruno César Rodrigues e Giulia Crippa. 4. A recuperação da informação e o conceito de informação: o que é relevante em mediação cultural? *Perspectivas em Ciência da Informação*, 16:45 – 64, 03 2011.

[Rep] Repositório do projeto. https://github.com/fabiokaspar/TCC. 19

[Vis] Visite São Paulo. https://web.archive.org/web/20150929081059/http://www.visitesaopaulo.com/dados-da-cidade.asp. Accessed: 2015-11-19. 1

[XML] Artigo sobre XML. https://web.archive.org/web/20150906111445/https://pt.wikipedia.org/wiki/XML. Accessed: 2015-11-27. 15