VERIFICAÇÃO DE UNICIDADE DE URLS EM COLETORES DE PÁGINAS WEB

WALLACE FAVORETO HENRIQUE

VERIFICAÇÃO DE UNICIDADE DE URLS EM COLETORES DE PÁGINAS WEB

Dissertação apresentada ao Programa de Pós-Graduação em Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais como requisito parcial para a obtenção do grau de Mestre em Ciência da Computação.

ORIENTADOR: NIVIO ZIVIANI

Belo Horizonte

Março de 2011

Henrique, Wallace Favoreto

H519v Verificação de unicidade de URLs em coletores de páginas web / Wallace Favoreto Henrique. — Belo Horizonte, 2011.
xviii, 45 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais. Departamento de Ciência da Computação.

Orientador: Nivio Ziviani.

1. Computação - Teses. 2. Recuperação da Informação - Teses. I. Orientador. II. Título.

CDU 519.6*73 (043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO

FOLHA DE APROVAÇÃO

Verificação de unicidade de URLs em coletores de páginas Web

WALLACE FAVORETO HENRIQUE

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. NIVIO ZIVIANI - Orientador
Departamento de Ciência da Computação - UFMG

PROF. EDLENO SILVA DE MOURA
Departamento de Ciência da Computação - UFAM

PROF. MARCO ANTÔNIO PINHEIRO DE CRISTO
Departamento de Ciência da Computação - UFAM

Belo Horizonte, 10 de março de 2011.

À minha família.

Agradecimentos

Em primeiro lugar, agradeço a Deus por estar sempre ao meu lado em todos os momentos da minha vida.

À minha família, meu pai, minha mãe, meu irmão e minha irmã, por todo amor, carinho, compreensão e sacrifício. Sem eles eu não conseguiria concluir esta dissertação.

Aos amigos da república Los Computeros, Alan, Anísio, Rickson e Tupy, por terem me aturado esses dois anos.

Aos amigos do LATIN e LBD, Cristiano, Fabiano, Guilherme, Tinti, Thales, Thiago e Wladmir, pelo companheirismo diário e por todo conhecimento que adquiri com eles.

Aos amigos de sempre, companheiros da Escola Agrotécnica Federal de Alegre-ES e da Universidade Federal do Espírito Santo, pelo apoio constante, mesmo à distância.

Aos amigos de infância e amigos de Muniz Freire, por me ajudarem a sonhar.

Ao Prof. Nivio Ziviani, por ter orientado a realização deste trabalho e pelos diversos valores que ele me ensinou.

Aos demais professores que me ajudaram a desenvolver esta dissertação, Edleno, Marco Cristo, Marcos Gonçalves, Jussara e Altigran. Muito obrigado por todas críticas e sugestões.

"Não temas o progresso lento,
receie apenas ficar parado."
(Provérbio Chinês)

Resumo

Uma das principais dificuldades existentes no desenvolvimento de um coletor de páginas web está no componente verificador de unicidade de URLs, pois estruturas de dados complexas são exigidas para garantir que a identificação das URLs ainda não coletadas seja feita de forma eficaz e eficiente. Caso o verificador de unicidade de URLs não ofereça eficácia e eficiência, os outros componentes do coletor serão prejudicados. Neste trabalho, apresentamos um novo algoritmo para verificar unicidade de URLs chamado VEUNI (VErificador de UNIcidade de URLs). O algoritmo VEUNI foi comparado com o melhor algoritmo conhecido na literatura, o qual foi considerado um *baseline* nos experimentos. O estudo comparativo entre o algoritmo VEUNI e o algoritmo *baseline* foi realizado por meio de uma simulação de uma coleta de aproximadamente 350 milhões de páginas, utilizando uma coleção de referência chamada ClueWeb09. Os resultados experimentais mostram que o algoritmo proposto é uma alternativa que pode ser utilizada com êxito em coletores de páginas que visam ser escaláveis para toda a *Web*.

Palavras-chave: Recuperação de Informação, Máquinas de Buscas, Coletores de Páginas da Web, Unicidade de URLs.

Abstract

One of the main difficulties in the development of a web crawler is in the component for verifying URL uniqueness, since complex data structures are required to ensure that the identification of URLs still not collected will be performed effectively and efficiently. If the component for verifying URL uniqueness is not effective and efficient, the performance of the other web crawler components will be affected. In this work we present a new algorithm for verifying URLs uniqueness, referred to as VEUNI (VErificador de UNIcidade de URLs). The algorithm VEUNI was compared with the best known algorithm in the literature, which was considered a baseline in the experiments. The comparative study between the algorithm VEUNI and the baseline was performed through a simulation of a collection of approximately 350 million pages, using a reference collection called ClueWeb09. Experimental results show that the proposed algorithm is an alternative that can be successfully used in web crawlers designed to be scalable to the entire Web.

Keywords: Information Retrieval, Search Engines, Web Crawlers, URL uniqueness.

Sumário

Capítulo 1

Introdução

1.1 Motivação

As máquinas de busca são fundamentais para os usuários da internet, visto que a quantidade e diversidade de informação disponível na Web dificulta a localização de itens de interesse via navegação. O usuário utiliza as máquinas de buscas para buscar informação sobre um tópico presente em um ou mais sites (consultas informacionais), para encontrar um site específico (consultas navegacionais) ou ainda para encontrar sites onde é possível realizar transações como compras ou *downloads* (consultas transacionais) [Broder, 2002].

Uma máquina de busca tradicional da Web possui três componentes básicos: coletor, indexador e processador de consultas. O coletor é responsável por encontrar, coletar e armazenar as páginas da Web. O indexador é responsável por criar um índice que permite a recuperação das páginas armazenadas. Já o processador de consultas é responsável por recuperar do índice as páginas mais relevantes de acordo com a consulta do usuário.

O desenvolvimento de um coletor de páginas Web é uma tarefa complexa, pois a partir de um conjunto inicial de URLs (semente), o coletor deve conseguir coletar páginas Web que possam interessar aos diversos usuários da máquina de busca. Como os usuários possuem interesses distintos, o coletor deve ser escalável para toda Web, pois um número significativo de páginas devem ser coletadas para para cobrir os interesses dos usuários.

Os quatro principais componentes de um coletor de páginas da Web são:

- *Fetcher*: responsável por realizar *download* de páginas da Web.
- Extrator de URLs: responsável por extrair as URLs existentes nas páginas coletadas.

- Verificador de Unicidade de URLs: responsável por informar quais URLs foram coletadas e quais URLs ainda precisam ser coletadas.

- Escalonador: responsável por encaminhar um conjunto de URLs ainda não coletadas ao *fetcher*.

Uma das principais dificuldades existentes no desenvolvimento de um coletor está no componente verificador de unicidade de URLs, pois estruturas de dados complexas são exigidas para garantir que a identificação das URLs ainda não coletadas seja feita de forma eficaz e eficiente. Para conseguir eficácia, é necessário que uma propriedade seja garantida: a resposta à consulta para saber se uma URL já foi coletada deve ser precisa. Se essa propriedade não for garantida as seguintes situações podem ocorrer:

1. Páginas ainda não coletadas, serão ignoradas e, portanto, não coletadas.

2. Páginas já coletadas, serão coletadas novamente.

Os dois problemas apontados causam desperdício de espaço de armazenamento, redução do número de páginas distintas coletadas por unidade de tempo, desperdício de banda de internet no *fetcher* e redução da capacidade de expansão da coleta, entre outros.

Além do verificador de unicidade de URLs ser eficaz, ele deve realizar suas funções de forma eficiente para não retardar o funcionamento do restante do coletor. Em um primeiro momento, a dificuldade de construir um verificador de unicidade de URLs pode ser subestimada. Por exemplo, a utilização de um gerenciador de banco de dados simplifica a gerência das páginas coletadas. Entretanto, testes realizados com gerenciadores de banco de dados mostram que o tempo requerido para a verificação de unicidade das URLs se torna um gargalo do funcionamento do coletor.

O principal objetivo desta dissertação é apresentar um novo sistema para verificar a unicidade de URLs em coletores de páginas Web. O sistema proposto é chamado VEUNI (VErificador de UNIcidade de URLs). Além de tratar de um problema desafiador no contexto de coletores de máquinas de busca de páginas da Web, este trabalho é importante para o projeto de construção de uma máquina de busca para a Web brasileira, em desenvolvimento dentro do Instituto Nacional de Ciência e Tecnologia para a Web (InWeb), sediado no Departamento de Ciência da Computação da Universidade Federal de Minas Gerais. O coletor em desenvolvimento no InWeb pretende ser capaz de lidar com centenas de milhões de páginas da Web. Uma coleta dessa magnitude é importante para a linha de pesquisa que esta desenvolvendo a máquina de busca e também para as outras linhas de pesquisa do InWeb.

1.2 Trabalhos Relacionados

Na literatura há poucos trabalhos que descrevem como construir um verificador de unicidade de URLs de um coletor de páginas Web. Dentre os trabalhos encontrados, selecionamos os mais importantes para ilustrar o que já foi pesquisado sobre esse tema.

O algoritmo proposto por [Pinkerton, 1994] utiliza um banco de dados (árvore-B) para verificar a unicidade de URLs. A vantagem dessa abordagem é a simplicidade, pois há inúmeros gerenciadores de banco de dados que podem ser utilizados para esse fim. Como desvantagem, podemos citar a ineficiência inerente ao uso de um SGBD que foi projetado para acesso a qualquer tipo de dados. Quando aplicado ao problema específico de encontrar URLs, o custo de acesso por URL se mostra muito alto.

O algoritmo Mercator-A [Heydon & Najork, 1999] difere do algoritmo de Pinkerton pela utilização de cache de memória. Quando uma URL não é encontrada no cache de memória, um acesso ao disco é realizado para obter a URL desejada. Em cada acesso ao disco para obter uma URL, um conjunto extra de URLs é retornado, no intuito de diminuir a necessidade de acessos ao disco. É interessante recuperar um conjunto de URLs em cada acesso ao disco para aumentar a probabilidade de que a próxima URL a ser consultada já esteja no cache de memória. O problema dessa abordagem é que, no pior caso, o método necessita de um acesso ao disco por URL e possui uma sobrecarga de leitura de um conjunto de URLs a cada consulta.

Os algoritmos Mercator-B [Najork & Heydon, 2001] e Polybot [Shkapenyuk & Suel, 2002] utilizam uma abordagem conhecida como *batch disk check* que consiste em acumular as URLs em uma área de armazenamento (*buffer*) na memória. Assim que a área de armazenamento encher, as URLs da memória são ordenadas e intercaladas com as URLs já conhecidas que estão armazenadas no disco. Entretanto, essa abordagem se torna ineficiente para grandes quantidades de dados, visto que o arquivo com todas as URLs precisa ser reescrito no disco frequentemente.

Uma proposta mais recente, conhecida como DRUM (*Disk Repository with Update Management*), pode ser encontrada no coletor IRLBot [Lee et al., 2009]. O algoritmo DRUM combina idéias dos algoritmos citados anteriormente, utilizando memória RAM e disco rígido em uma estratégia conhecida como *bucket sort*. Os autores relataram a coleta de 6 bilhões de páginas HTML em 41 dias com esse algoritmo. Além disso, os autores compararam o algoritmo DRUM com outros algoritmos e verificaram que o desempenho obtido com o algoritmo DRUM foi superior aos demais. Por isso, o algoritmo DRUM pode ser considerado o estado-da-arte em verificação de unicidade de URLs. Por essa razão realizamos um estudo comparativo entre a nossa proposta e o algoritmo DRUM. A descrição detalhada do algoritmo DRUM pode ser encontrada no Capítulo 4.

1.3 Objetivos

O objetivo principal desta dissertação é propor um novo sistema para verificar a unicidade de URLs em coletores de páginas web. Como o sistema proposto é parte de um coletor de páginas da Web, é importante que haja uma discussão sobre a arquitetura do coletor. Para verificar a qualidade do sistema proposto, o melhor algoritmo conhecido na literatura para verificar unicidade de URLs foi utilizado, sendo considerado um *baseline* para os experimentos realizados. O objetivo no caso é realizar um estudo comparativo entre o sistema proposto e o algoritmo escolhido da literatura.

1.4 Contribuições

A principal contribuição desta dissertação é a especificação, projeto e implementação de um novo sistema verificador de unicidade de URLs, chamado VEUNI (VErificador de UNIcidade de URLs). O sistema VEUNI foi utilizado dentro de um coletor real de páginas web. Além disso, resultados experimentais obtidos na utilização do sistema VEUNI sobre uma coleção de aproximadamente 350 milhões de URLs mostram que ele é mais eficiente em tempo de processamento que o algoritmo considerado como *baseline* nos experimentos.

As contribuições específicas deste trabalho são:

1. Discussão da arquitetura do coletor de páginas da Web em que o sistema VEUNI está inserido (Capítulo 2);

2. Descrição detalhada do sistema VEUNI (Capítulo 3);

3. Discussão de propostas que levaram ao sistema VEUNI, com o objetivo de apresentar os problemas de cada alternativa (Seção 3.3);

4. Apresentação do algoritmo utilizado como *baseline* no estudo comparativo com o sistema VEUNI (Seção 4.1)

5. Criação de um ambiente para experimentação dos algoritmos estudados neste trabalho. O ambiente foi criado para avaliar o desempenho do sistema VEUNI e do algoritmo escolhido como *baseline* sem a necessidade de coletar páginas diretamente da Web. Para isso foi utilizada a coleção ClueWeb09[1], que contém aproximadamente 1 bilhão de páginas da Web. Essa coleção foi criada e é distribuída pela Universidade de Carnegie Mellon (Seção 4.2).

[1]ClueWeb09: http://boston.lti.cs.cmu.edu/Data/clueweb09/

1.5 Organização da Dissertação

Essa dissertação está estruturada em cinco capítulos, dos quais este é o primeiro. O Capítulo 2 apresenta uma arquitetura para coletores de páginas web e descreve seus principais componentes. O Capítulo 3 apresenta o sistema VEUNI para verificação de unicidade de URLs. O Capítulo 4 apresenta um estudo comparativo do sistema VEUNI com o algoritmo considerado o estado-da-arte na literatura. Finalmente, o Capítulo 5 apresenta as conclusões e os trabalhos futuros.

Capítulo 2

Arquitetura do Coletor de Páginas Web

O coletor de páginas web é o componente da máquina de busca responsável por encontrar os documentos que irão compor o conjunto resposta a uma consulta realizada por um usuário. Consequentemente, se os documentos potencialmente mais relevantes da Web não forem previamente coletados, se torna mais difícil retornar aos usuários resultados que lhes interessem.

Um problema importante é descobrir quais são os documentos potencialmente relevantes da Web, visto que cada usuário da máquina de busca possui um interesse peculiar. Uma possível solução para esse problema é coletar o maior número possível de páginas da Web. Por outro lado, ao se coletar muitas páginas, o número de documentos de baixa qualidade encontrados na coleta será maior. Esse fato dificulta o trabalho do processador de consultas, visto que será mais complexo encontrar os melhores documentos a serem apresentados aos usuários. Entretanto, é dessa maneira que uma máquina de busca comumente funciona: os coletores coletam o que for possível e os processadores de consulta são os responsáveis por selecionar os melhores documentos coletados.

Os principais componentes de um coletor de páginas web são: *fetcher*, extrator de URLs, verificador de unicidade de URLs e escalonador. As seções a seguir apresentam cada um desses componentes.

2.1 *Fetcher*

Um coletor de páginas web precisa de um componente responsável por coletar o conjunto de páginas que poderá ser utilizado nas respostas às consultas dos usuários,

conhecido como *fetcher*. O principal objetivo do *fetcher* é coletar páginas da Web a partir de um conjunto de URLs fornecido como entrada, conforme mostra a Figura 2.1.

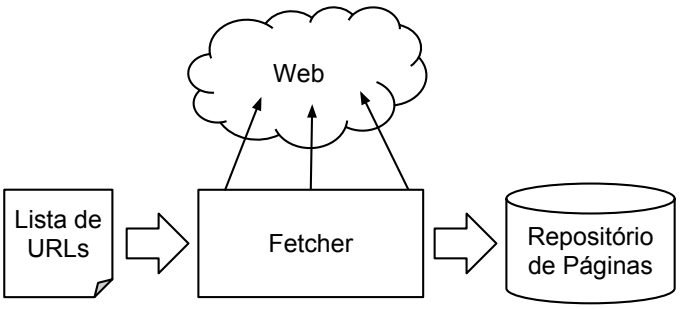

Figura 2.1. Entrada e saída do *fetcher*.

O *fetcher* encontra a localização de uma página da Web utilizando um identificador único, conhecido como URL (*uniform resource locator*). Uma URL é composta de três componentes: o método de acesso, o nome do servidor e o caminho. Os três componentes de uma URL são ilustrados na Figura 2.2.

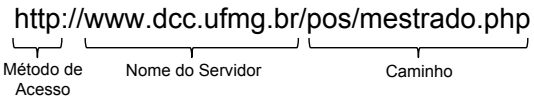

Figura 2.2. Componentes de uma URL.

O método de acesso de uma URL indica o protocolo que deve ser utilizado para acessar a página referente a URL. A maioria das páginas web são armazenadas em servidores web que utilizam o protocolo HTTP (*Hypertext Transfer Protocol*) para realizaram a troca de informação com algum cliente. O nome do servidor em uma URL representa o nome do computador que hospeda o servidor web responsável pela página referente a URL. Por último, o caminho de uma URL é o identificador utilizado para localizar a página web dentro do servidor.

O *download* de uma página da Web utilizando sua URL é realizada seguindo uma sequência de etapas. Primeiramente, é necessário que o *fetcher* se conecte a um servidor de nomes (DNS). O servidor de nomes é utilizado para traduzir o nome do servidor presente na URL para um endereço IP (*internet protocol*). O endereço IP normalmente é um número de 32 bits, entretanto, algumas redes já utilizam endereços IP de 128 bits. Após identificar o endereço IP, a conexão com o servidor que hospeda a página pode ser estabelecida. Por convenção, quando se trata de uma requisição a páginas web, essa conexão é realizada utilizando a porta 80.

Uma vez que a conexão estiver estabelecida, o *fetcher* envia uma requisição HTTP ao servidor para recuperar uma página. O tipo de requisição HTTP normalmente utilizado para se recuperar páginas é o GET. O GET requisita a página ao servidor e o servidor envia a página ao *fetcher* utilizando as especificações do protocolo HTTP.

A resposta do servidor pode ser dividida em duas partes: cabeçalho HTTP e conteúdo da página. O cabeçalho HTTP é retornado primeiramente ao *fetcher*. Nele é possível encontrar algumas características da página requisitada, tais como, codificação da página, tipo do documento, tamanho em bytes da página e a data que a página foi requisitada, dentre outras.

Uma técnica normalmente utilizada para aumentar a eficiência do *fetcher* consiste de utilizar múltiplas *threads* na realização da coleta. Isso se faz necessário, tendo em vista que o *fetcher* passa muito tempo apenas esperando pela resposta a uma requisição realizada: ele espera pela resposta do servidor DNS, espera pelo estabelecimento da conexão com o servidor que hospeda a página e espera pelo *download* da página do servidor. Vale ressaltar que a construção de um *fetcher* é mais difícil quando um dos requisitos da especificação é coletar um grande conjunto de páginas em um pequeno intervalo de tempo.

Com o *fetcher* utilizando múltiplas *threads* para coletar as páginas, uma preocupação que surge é a de não sobrecarregar com muitas requisições os servidores acessados. Para evitar esse problema, os coletores devem respeitar uma convenção conhecida como *politeness*. O *politeness* aconselha os coletores a não coletar mais de uma página por vez de um determinado servidor e esperarem pelo menos alguns segundos, ou minutos, para fazer uma nova requisição a um servidor já visitado [Boldi et al., 2004].

O desrespeito ao *politeness* pode levar a implicações indesejáveis ao servidor visitado, tais como: lentidão aos usuários que acessarem as páginas do servidor devido ao consumo exagerado de recursos (como memória ou processamento por exemplo), geração de *logs* que não refletem o acesso ao servidor por parte dos usuários e até mesmo tornar os recursos do sistema indisponíveis para seus utilizadores. Em suma, o acesso ao servidor por parte do coletor de páginas da Web pode ser considerado um ataque de negação de serviço (*Denial of Service*).

Além de gerar consequências indesejáveis ao servidor coletado, o desrespeito ao *politeness* pode gerar prejuízo à qualidade da coleta. Por exemplo, o responsável pelo servidor visitado de forma indevida pode bloquear o acesso às suas páginas. Caso haja um bloqueio como esse, não será mais possível coletar páginas do servidor bloqueado.

Outro ponto que merece destaque dentro da arquitetura do *fetcher* é o protocolo de exclusão *robots* (*Robots Exclusion Protocol*). O protocolo de exclusão *robots*, tam-

bém conhecido como protocolo `robots.txt`, é uma convenção utilizada para indicar quais páginas públicas de um site não devem ser coletadas pelos coletores. Se o proprietário de um site deseja informar quais páginas ele permite que sejam coletadas por coletores de páginas da Web, ele deve deixar disponível no diretório raiz do seu site um arquivo texto chamado `robots.txt`. Nesse arquivo o proprietário do site irá indicar, utilizando um formato específico, as permissões de acesso ao site pelos coletores. Abaixo dois exemplos de conteúdo de um arquivo `robots.txt` são mostrados.

- Permite que os coletores Web coletem todas as páginas do site:

```
----------------
User-agent: *
Disallow:
---------------
```

- Proíbe a entrada dos coletores Web no diretório /private/ e no diretório /data/.

```
----------------
User-agent: *
Disallow: /private/
Disallow: /data/
---------------
```

Os coletores de páginas da Web que seguem as instruções do protocolo primeiramente acessam as informações do arquivo `robots.txt` antes de acessar qualquer outra página pertencente ao site para obterem os diretórios em que o acesso é negado. Se o arquivo `robots.txt` não existir, os coletores assumem que o proprietário do site não deseja privar nenhuma página da coleta.

Após coletar as páginas correspondentes ao conjunto de URLs passado como entrada, o *fetcher* deve armazenar as páginas coletadas de tal forma que seja permitido acessá-las de forma eficiente quando necessário. Na arquitetura do coletor, o acesso as páginas armazenadas é requisitado pelo componente extrator de URLs.

Tendo em vista que é grande o número de páginas coletadas pelos coletores, não é recomendado utilizar um sistema gerenciador de banco de dados para armazená-las. Experimentalmente observamos que a forma mais adequada de se armazenar as páginas coletadas consiste de fazer o armazenamento de forma sequêncial no disco. Isso pode ser feito criando um arquivo chamado $Repositorio\_de\_Paginas$ e salvando as páginas nesse arquivo à medida que elas são coletadas.

2.2 Extrator de URLs

O extrator de URLs é o componente responsável por extrair as URLs existentes dentro das página coletadas. A entrada e saída do componente extrator de URLs pode ser visualizada na Figura 2.3. Esse componente requisita como entrada um conjunto de páginas HTML e produz como saída três arquivos: $UrlsColetadas$, $UrlsExtraidas$ e $MetadadosUrlsColetadas$.

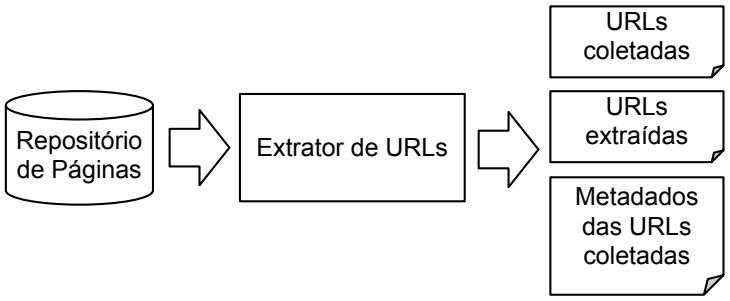

Figura 2.3. Entrada e saída do extrator de URLs.

O conjunto de páginas fornecido como entrada desse componente é obtido por meio de uma leitura sequêncial das páginas armazenadas no $Repositorio\_de\_Paginas$. O arquivo $UrlsColetadas$ conterá o conjunto de URLs coletadas no ciclo de coleta corrente, o arquivo $UrlsExtraidas$ conterá o conjunto de URLs extraídas das páginas coletadas e o arquivo $MetadadosUrlsColetadas$ conterá os metadados que auxiliam um posterior acesso às páginas coletadas.

Os principais metadados armazenados em $MetadadosUrlsColetadas$ são: localização no disco do $Repositorio\_de\_Paginas$ referente à página coletada e o $offset$ para localizar a página dentro do $Repositorio\_de\_Paginas$. É importante armazenar esses metadados, pois o acesso ao conteúdo das páginas coletadas é necessário por outros componentes da máquina de busca tais como, o Indexador e o Detector de Réplicas.

A extração de URLs de uma página usualmente é realizada utilizando um analisador sintático HTML (*parser* HTML). Um analisador sintático genérico analisa uma sequência de entrada no intuito de determinar sua estrutura gramatical segundo uma determinada gramática formal. A gramática formal utilizada por um analisador sintático HTML corresponde as regras sintáticas utilizadas para se programar na linguagem HTML.

Com a utilização de um analisador sintático HTML, a identificação e extração de URLs de uma página é simplificada. As URLs são identificadas procurando pela *tag*

HTML que indica a existência de um *link* (`<a>`) na página. A seguir é mostrado um exemplo de página HTML e as URLs nela encontradas.

- Exemplo de página HTML:

```
<html>
<body>
<a href="http://www.ufmg.br">Universidade Federal de Minas Gerais</a>
<a href="http://www.cnpq.br">CNPq</a>
<a href="http://www.capes.gov.br">CAPES</a>
</body>
</html>
```

- URLs extraídas:

```
http://www.ufmg.br
http://www.cnpq.br
http://www.capes.gov.br
```

Apesar da descrição e do funcionamento deste componente ser simples, o extrator de URLs tem um papel fundamental dentro da arquitetura do coletor. Ele é o componente responsável por descobrir as centenas de milhares de páginas que serão coletadas.

Uma maneira de otimizar o desempenho do extrator de URLs é delegar a mais de um processo a tarefa de extrair as URLs existentes nas páginas coletadas. Para isso é necessário atribuir a cada processo extrator de URLs um subconjunto das páginas armazenadas no $Repositorio\_de\_Paginas$. Vale ressaltar que é imprescindível que os processos criados gerem arquivos de saída independentes, para evitar a concorrência no processo de escrita desses arquivos.

Após o término da execução dos processos, os arquivos de saída gerados por cada um deles devem ser agrupados. Desta forma, esses arquivos passam a ser representados em apenas três: $UrlsColetadas$, $UrlsExtraidas$ e $MetadadosUrlsColetadas$. Esses três arquivos são a entrada do componente verificador de unicidade de URLs, que será descrito a seguir.

2.3 Verificador de Unicidade de URLs

O verificador de unicidade de URLs requisita como entrada dois arquivos de URLs e um arquivo com os metadados das páginas coletadas. Esses três arquivos são gerados

pelo componente extrator de URLs. O primeiro arquivo é chamado $UrlsColetadas$ e possui as URLs coletadas no ciclo de coleta. O segundo arquivo é chamado $UrlsExtraidas$ e possui as URLs extraídas no ciclo de coleta. O terceiro arquivo é chamado $MetadadosUrlsColetadas$ e possui os metadados das páginas coletadas. A saída produzida pelo verificador de unicidade de URLs é um repositório que armazena de forma unívoca as URLs encontradas na coleta e os metadados das páginas coletadas. A Figura 2.4 ilustra as entradas e a saída do verificador de unicidade de URLs. Os objetivos específicos desse componente são:

i. Armazenar em um repositório de URLs (R) o conjunto de URLs conhecidas pelo coletor.

ii. Armazenar informações sobre o armazenamento das páginas web referentes às URLs coletadas.

iii. Evitar que uma URL já coletada seja coletada novamente.

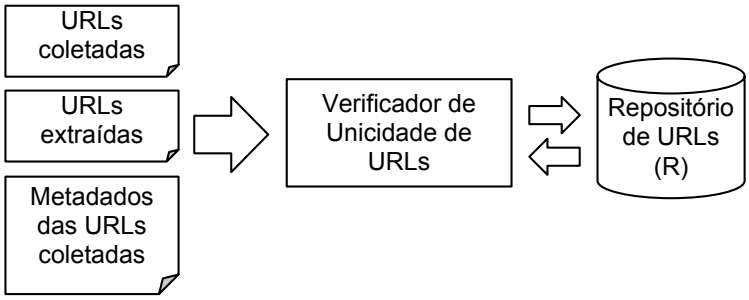

Figura 2.4. Entrada e saída do verificador de unicidade de URLs.

Para atingir esses objetivos, o verificador de unicidade de URLs analisa as URLs existentes nos arquivos $UrlsExtraidas$ e $UrlsColetadas$ gerados pelo extrator de URLs no intuito de identificar os seguintes conjuntos:

- Conjunto 1: URLs que já foram coletadas em ciclos anteriores.
- Conjunto 2: URLs que o coletor descobriu em ciclos anteriores, mas que ainda não foram coletadas.
- Conjunto 3: URLs coletadas no ciclo corrente.
- Conjunto 4: Novas URLs.

Após identificar os conjuntos acima mencionados, o verificador de unicidade de URLs descarta as URLs pertencentes ao conjunto 1, visto que são URLs referentes a páginas que já foram anteriormente coletadas pelo coletor. Essa mesma ação é tomada com as URLs pertencentes ao conjunto 2, pois mesmo que essas URLs ainda não tenham sido coletadas, não é necessário armazená-las novamente em R.

Já em relação as URLs coletadas no ciclo corrente (conjunto 3), o verificador de unicidade de URLs atualiza em R as informações de armazenamento das páginas web referentes a essas URLs indicando que elas foram coletadas. As informações sobre o armazenamento das páginas coletadas são recuperadas do arquivo $MetadadosUrlsColetadas$. Por último, o conjunto 4, composto de novas URLs, é incorporado a R.

2.4 Escalonador

O escalonador é o componente responsável por selecionar do repositório de URLs R o conjunto de URLs que será coletado no próximo ciclo do coletor. Vale ressaltar que o componente que mantém o repositório R atualizado é o verificador de unicidade de URLS. Após o escalonador selecionar o conjunto de URLs, ele cria o arquivo $Lista\_de\_URLs$, que contém as URLs que devem ser encaminhadas ao *fetcher*, conforme ilustra a Figura 2.5. A seguir iremos descrever como o conjunto de URLs pode ser selecionado dentro do repositório R.

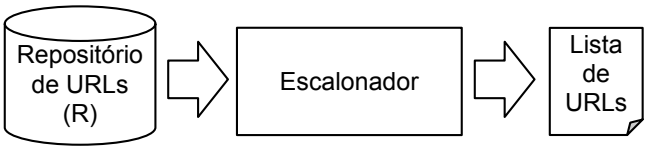

Figura 2.5. Entrada e saída do escalonador.

O componente escalonador possui acesso a todas as URLs existentes em R. Entretanto, as URLs que interessam para ele são apenas as que ainda não foram coletadas. Este fato é o que motiva armazenar no repositório R um metadado dizendo se uma dada URL já foi ou não coletada. Portanto, esse é o universo de URLs que o escalonador considera nos escalonamentos e iremos chamar esse novo conjunto de $URLsCandidatas$.

Após identificar o conjunto $URLsCandidatas$, o escalonador deve adotar alguma política para selecionar, dentre esse conjunto, as URLs que serão passadas ao *fetcher*.

Usualmente é definido uma constante $N_{escalonamento}$ que indica o número de URLs que o *fetcher* receberá em cada ciclo. Dentre as políticas existentes, podemos citar as seguintes: FIFO (*First In First Out*), *PageRank* [Page et al., 1998], número de páginas do servidor e tempo médio de resposta do servidor [Coffman et al., 1997]. A seguir descrevemos superficialmente as políticas mencionadas.

Na política de seleção baseada no algoritmo FIFO, as URLs que forem conhecidas primeiro, serão encaminhadas para o *fetcher*. A vantagem desta política é a simplicidade de implementação. Já a desvantagem é a alta probabilidade da expansão da coleta ser prejudicada. A capacidade da coleta expandir é afetada, pois muitas URLs de apenas um determinado site poderão ser coletadas antes de um novo site ser coletado.

Uma política mais adequada é a que considera o *PageRank* das páginas. O *PageRank* consiste de uma métrica que indica a importância de uma página em relação às outras páginas da Web. No cálculo do *PageRank* de uma página p, é considerado o número de páginas que apontam para p e o valor do *PageRank* dessas páginas. Quanto mais páginas apontarem para a página p e quanto mais importantes forem essas páginas, maior será o *PageRank* da página p [Page et al., 1998]. Utilizando o *PageRank*, é possível selecionar as $N_{escalonamento}$ URLs considerando, por exemplo, apenas os $N_{servidores}$ com *PageRank* mais alto.

Outra possível política que pode ser adotada considera o número de páginas dos servidores que serão escalonados. Essa política prioriza os servidores com um maior número de páginas e é justamente esses sites que usualmente são os mais importantes para os usuários. Entretanto, há sites com poucas páginas que também são importantes para os usuários de uma máquina de busca e deveriam também ser escalonados para serem coletados. Portanto, essa política prioriza os grandes servidores, mas tem a desvantagem de penalizar os servidores com poucas páginas.

No intuito de obter uma taxa de coleta mais alta, uma política de escalonamento baseada na velocidade de resposta dos servidores escalonados pode ser utilizada. Essa política constrói um *ranking* de servidores utilizando o tempo médio necessário para realizar o *download* das páginas de cada servidor. No topo deste *ranking* iremos encontrar os servidores que respondem mais rapidamente e na base do *ranking* encontraremos os servidores mais lentos. Com o *ranking* gerado, uma proporção pode ser utilizada para indicar o número de servidores rápidos que serão escalonados para cada servidor lento escalonado.

Capítulo 3

Sistema Proposto para Verificação de Unicidade de URLs

Este capítulo apresenta um novo sistema para verificar unicidade de URLs em um coletor de páginas da Web chamado VEUNI - VErificador de UNIcidade de URLs. Toda URL extraída de uma página recém coletada tem que ser verificada pelo sistema VEUNI se já foi coletada anteriormente ou não. O sistema para verificar a unicidade de URLs é um dos componentes mais importantes de um coletor de páginas da Web.

A Seção 3.1 apresenta a descrição do sistema proposto. Antes da apresentação propriamente dita do sistema VEUNI são discutidos o contexto de funcionamento do coletor de páginas da Web e a natureza do problema a ser resolvido. A Seção 3.2 apresenta um refinamento dos principais componentes do sistema VEUNI. A Seção 3.3 apresenta versões anteriores do sistema VEUNI com o objetivo de discutir os principais problemas encontrados que levaram ao insucesso de cada uma delas.

3.1 Descrição do Sistema VEUNI

O sistema para verificar unicidade de URLs é um dos componentes de um coletor de páginas da Web. Uma descrição detalhada da arquitetura de um coletor de páginas da Web pode ser vista no Capítulo 2. Um coletor de páginas da Web é constituído de quatro componentes principais, a saber: (i) *fetcher* - realiza a coleta de um conjunto de páginas web referentes a uma lista de URLs fornecida como entrada; (ii) extrator de URLs - extrai as URLs existentes dentro das páginas coletadas; (iii) verificador de unicidade de URLs - atualiza o repositório de URLs com as URLs coletadas e adiciona novas URLs encontradas; (iv) escalonador - seleciona do repositório um conjunto de URLs que ainda não foi coletado e encaminha esse conjunto ao *fetcher*.

A Figura 3.1 apresenta o ciclo de coleta envolvendo os quatro componentes de um coletor. No passo 1, o *fetcher* recebe do escalonador um conjunto de URLs. Para cada URL, o *fetcher* coleta a página no servidor que está indicado na URL (vide Seção 2.1 para descrição dos componentes de uma URL). No passo 2, o extrator de URLs extrai as URLs de cada página trazida pelo *fetcher*. No passo 3, o verificador de unicidade de URLs verifica se cada URL já foi coletada. No passo 4, o escalonador escolhe um novo conjunto de URLs que deve ser passado para o *fetcher*, fechando o ciclo de coleta.

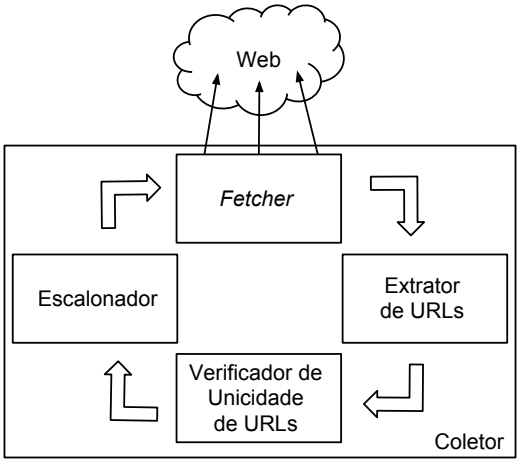

Figura 3.1. Ciclo de coleta de páginas da Web.

O verificador de unicidade de URLs deve garantir que uma mesma URL não seja submetida para o escalonamento mais de uma vez em um ciclo de coleta. Uma solução ingênua para resolver este problema é apresentada no Programa 1. A entrada do Programa 1 consiste de um conjunto de URLs encontradas no ciclo de coleta. A saída do Programa 1 é o repositório R de URLs conhecidas pelo coletor atualizado com as URLs do ciclo de coleta.

Programa 1 Algoritmo ingênuo para verificação de unicidade de URLs.
Entrada: U: Conjunto de URLs encontradas no ciclo de coleta.
Saída: Repositório R de URLs existentes no coletor atualizado com as URLs de U.

```
1: para todo URL em U faça
2:    se URL está presente em R então
3:       URL é descartada.
4:    se não
5:       URL é adicionada a R.
```

Considere que: (i) o conjunto R é grande e está armazenado em disco; (ii) o número de URLs a serem verificadas em cada ciclo do coletor também é grande; (iii) o algoritmo do Programa 1 realiza uma busca no disco para cada URL a ser verificada.

A coleção utilizada nos experimentos contém aproximadamente 350 milhões de URLs e aproximadamente 1 milhão de URLs são verificadas a cada ciclo. Assim sendo, se desconsiderarmos o custo de acesso ao repositório, para um disco que tenha tempo de *seek* igual a 10 milissegundos, 1 milhão de acessos leva cerca de 3 horas de processamento. Em outras palavras, a cada ciclo do coletor a fase de verificação de unicidade de URLs leva, pelo menos, 3 horas.

Uma forma de melhorar o desempenho do algoritmo do Programa 1 é tratar todas as URLs em *batch*. Assim evitamos a busca individual de URLs no repositório R armazenado em disco e tratamos um bloco inteiro de URLs a cada iteração. A nova proposta de algoritmo que trata as URLs em *batch* é chamada sistema VEUNI. A seguir apresentamos uma discussão sobre a entrada e saída de dados do sistema VEUNI, conforme ilustra a Figura 3.2.

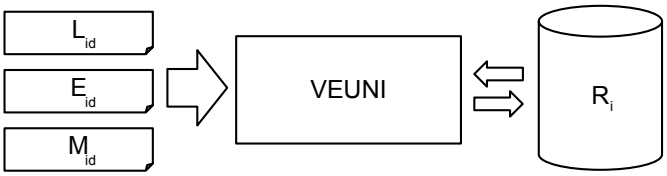

Figura 3.2. Entrada e saída do sistema VEUNI.

Como mostra a Figura 3.2, em cada ciclo de coleta identificado por id os três principais componentes de entrada do sistema VEUNI são:

- L_{id}: URLs relativas às páginas coletadas pelo *fetcher* no ciclo id;
- E_{id}: URLs extraídas das novas páginas coletadas no ciclo id;
- M_{id}: Metadados obtidos sobre as páginas coletadas, onde cada página corresponde a uma URL de L_{id}.

Para discutir a saída R_i do sistema VEUNI é necessário apresentar a estrutura de dados R. O repositório R contém todas as URLs conhecidas pelo coletor ao iniciar o ciclo de coleta id. Para evitar uma reescrita no disco de todo o conjunto R a cada ciclo de coleta, R é subdividido em blocos R_i, conforme ilustra a Figura 3.3. Neste caso, o repositório R é constituído de um vetor contendo N entradas, onde cada entrada R_i, $0 \leq i < N$, contém URLs relacionadas a um determinado conjunto de servidores web. No momento em que o extrator encontra uma nova URL, uma função *hash* é aplicada no nome do servidor para indicar para qual R_i a URL vai ser dirigida. Assim, cada entrada R_i contém informação sobre um conjunto de k servidores S_{ij}, $0 \leq i < N$ e

$0 \leq j < k$. Nos experimentos foram utilizados $k = 100.000$ servidores por entrada e para uma coleta envolvendo 1.000.000 de servidores, usamos $N \geq 10$.

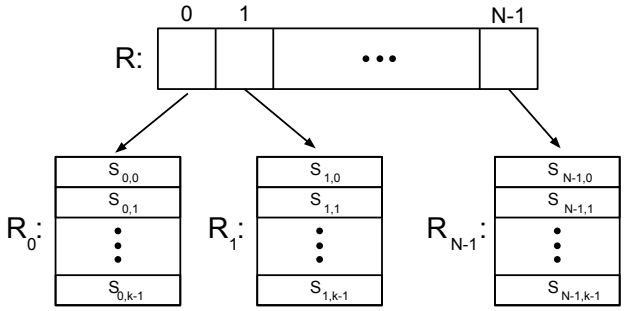

Figura 3.3. Estrutura de dados descrevendo o repositório R.

O Programa 2 apresenta o primeiro refinamento do sistema VEUNI. A seguir iremos descrever cada linha do sistema VEUNI e destacar os principais passos que diferenciam essa abordagem de outras utilizadas para resolver o problema de verificação de unicidade de URLs.

Programa 2 Primeiro refinamento do sistema VEUNI.

Entrada: id: Identificação do ciclo de coleta;
L_{id}: URLs coletadas no ciclo de coleta;
E_{id}: URLs extraídas no ciclo de coleta;
M_{id}: Metadados das páginas coletadas no ciclo de coleta.

Saída: Repositório R_i de URLs existentes no coletor atualizado com as URLs do ciclo de coleta id.

1: i recebe id mod N.
2: E_{id} recebe a união de E_{id} com AUX_i.
3: Particione L_{id} e E_{id} em subgrupos B_j, $0 \leq j < \text{Nserv}_{id}$, tal que B_j possua URLs de um mesmo servidor j.
4: Inicializa um novo repositório Rnovo_i em disco.
5: **para todo** S_{ij} em R_i **faça**
6: Intercala S_{ij} com B_j.
7: Grava em Rnovo_i o resultado da intercalação.
8: **para todo** B_j contendo URLs de novos servidores **faça**
9: Insere B_j em Rnovo_i.
10: Rnovo_i passa a ser o novo R_i.

O primeiro passo para o entendimento do sistema VEUNI é considerar que o escalonador utiliza um bloco de URLs R_i para selecionar as URLs que deverão ser repassadas ao *fetcher*. Na arquitetura de um coletor que utiliza o VEUNI, apenas o R_i selecionado pelo escalonador será atualizado no ciclo de coleta. O bloco R_i selecionado

pelo escalonador é identificado na linha 1 do sistema VEUNI por meio da utilização do número que indica o ciclo de coleta (id) e do número de blocos existentes N.

Para descrever o que é realizado na linha 2 do sistema VEUNI é importante destacar que no momento de extração das URLs que irão compor o conjunto de URLs extraídas E_{id}, uma função *hash* é utilizada para indicar os blocos R_i, $0 \leq i < N$ que cada URL extraída pertence. Desse modo, apenas as URLs extraídas que são mapeadas para o bloco $R_{id \bmod N}$ são utilizadas para formar o conjunto E_{id}. As URLs extraídas que são mapeadas para outros blocos R_i, $i \neq (id \bmod N)$, são armazenadas temporariamente em um arquivo chamado AUX_i, $0 \leq i < N$, $i \neq (id \bmod N)$. Por esse motivo, é necessário que o conjunto E_{id} seja enriquecido com as URLs que pertencem ao $R_{id \bmod N}$ extraídas em ciclos anteriores de coleta. Na linha 2 é realizado o enriquecimento do conjunto E_{id} por meio da união do conjunto E_{id} com o conjunto $\text{AUX}_{id \bmod N}$.

Na linha 3, o conjunto de URLs coletadas L_{id} e o conjunto de URLs extraídas E_{id} são particionados em B_j, $0 \leq j < \text{Nserv}_{id}$ (Nserv_{id} corresponde ao número de servidores encontrados no ciclo id de coleta), por meio de uma função *hash*, onde cada entrada da tabela *hash* possui uma lista encadeada de URLs relativas a cada servidor. O particionamento por nome de servidor das URLs de L_{id} e E_{id} é identificado no Programa 2 como passo de particionamento.

Na linha 4, um novo arquivo em disco chamado Rnovo_i é aberto para receber a intercalação das URLs encontradas no ciclo de coleta id com o o repositório R_i corrente de URLs existentes no coletor. No laço contendo as linhas 5, 6 e 7, para cada servidor S_{ij} de R_i ocorre a intercalação com B_j (passo de intercalação) e posterior gravação em Rnovo_i em disco do resultado da intercalação. No laço contendo as linhas 8 e 9, as URLs de novos servidores que estão em B_j são gravadas em Rnovo_i em disco (passo de inserção de novos servidores). Finalmente, na linha 10, R_i passa a apontar para Rnovo_i em disco.

A vantagem da proposta que acabamos de apresentar é que evitamos atualizar todos os blocos de URLs R_i em cada escalonamento e conseguimos manter constante o desempenho do sistema, independentemente do número de URLs existentes, pois o desempenho irá depender apenas do número de servidores (k) de cada R_i. O número k de servidores pode ser controlado. Caso um R_i atinja um número de servidores maior que k, um novo bloco R_N é criado e as URLs de cada R_i, $0 \leq i < N$, são redistribuídas entre todos os $N + 1$ blocos.

Os passos de particionamento (linha 3), intercalação (linha 6) e inserção de novos servidores (linha 9) são descritos em detalhes na Seção 3.2.

3.2 Refinamento do Sistema VEUNI

Esta seção descreve em maiores detalhes o sistema VEUNI. Em particular, mostramos como a representação das URLs é realizada (Seção 3.2.1), e detalhamos os passos de particionamento (Seção 3.2.2), intercalação (Seção 3.2.3) e inserção de novos servidores (Seção 3.2.4).

3.2.1 Representação das URLs Armazenadas

As URLs são armazenadas de forma ordenada em um arquivo em disco chamado URLs_i, conforme ilustra a Figura 3.4. É importante armazenar as URLs de forma ordenada, visto que a verificação de unicidade de URLs do sistema VEUNI é realizada por meio de uma série de intercalações de conjuntos ordenados. Como pode ser visto na Figura 3.4, no arquivo URLs_i são armazenados apenas os caminhos relativos a cada URL de um servidor.

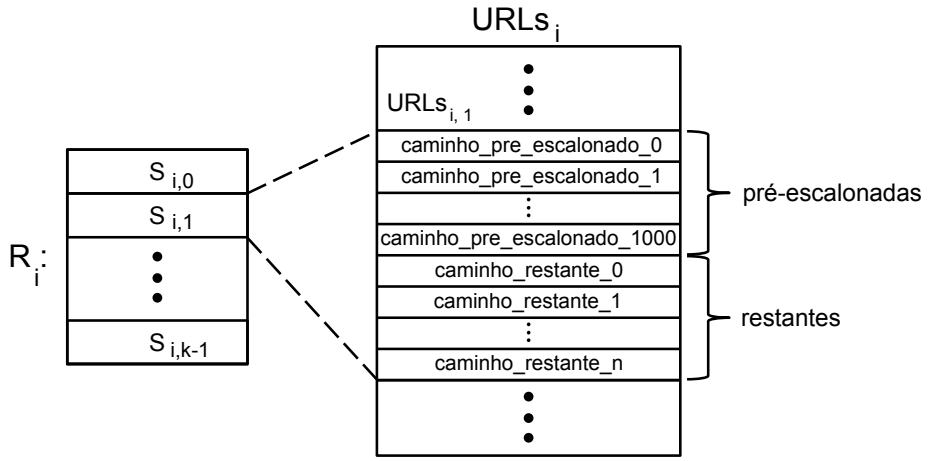

Figura 3.4. Estrutura de dados descrevendo um R_i.

Os caminhos relativos a cada URL de um servidor são divididos em duas categorias: *pré-escalonadas* e *restantes*. A categoria *pré-escalonadas* representa as URLs que podem ser utilizadas pelo escalonador na geração do escalonamento e a categoria *restantes* representa as URLs que serão escalonadas apenas quando todas as URLs da categorias *pré-escalonadas* já tiverem sido coletadas. Na Figura 3.4, as categorias *pré-escalonadas* e *restantes* podem ser visualizadas. As categorias *pré-escalonadas* e *restantes* são utilizadas com o objetivo de auxiliar o escalonador a selecionar as URLs que serão encaminhadas ao *fetcher*. Ao esgotar as URLs da categoria *pré-escalonadas*

novas URLs da categoria *restantes* são escolhidas para criar um novo conjunto *pré-escalonadas*. A escolha do novo conjunto *pré-escalonadas* obedece determinados critérios, tais como se é a primeira coleta da página, a data da última coleta para casos de refrescamento de páginas já coletadas, entre outros. Nos experimentos o tamanho da categoria *pré-escalonadas* foi considerado como 1.000 URLs.

Além de armazenar as URLs conhecidas de um conjunto de servidores, um URLs_i armazena um conjunto de informação (metadados) que auxiliam a recuperação das páginas web coletadas. Os metadados são utilizados também para armazenar informação referente à coleta realizada, tais como, momento que a página foi coletada e tempo necessário para realizar a coleta. É interessante armazenar essas informações, pois elas podem ser utilizadas pelo escalonador no momento de selecionar as próximas URLs a serem coletadas.

3.2.2 Particionamento de L_{id} e E_{id} em Subgrupos B_j

Esta subseção trata do particionamento do conjunto de URLs coletadas (L_{id}) e do conjunto de URLs extraídas (E_{id}) em subgrupos B_j, $0 \leq j < \text{Nserv}_{id}$, tal que B_j possua URLs de um mesmo servidor j, apresentado na linha 3 do Programa 2.

Há dois motivos que levam a realização do particionamento das URLs encontradas no ciclo de coleta. O primeiro deles diz respeito à maneira que as URLs são armazenadas nos blocos R_i. Como visto na Figura 3.3, cada bloco R_i é dividido em S_{ij}, $0 \leq i < N$ e $0 \leq j < k$. Desse modo, para criar um S_{ij}, primeiramente, as URLs do Servidor_{ij} devem ser agrupadas. Com as URLs agrupadas por servidor no bloco de URLs R_i, o escalonador de URLs consegue evitar que em um pequeno intervalo de tempo seja escalonado ao *fetcher* um número elevado de URLs de um mesmo servidor, respeitando assim o *politeness* (vide Seção 2.1 para maiores detalhes sobre *politeness*).

O segundo motivo para a realização do particionamento das URLs é auxiliar a etapa de verificação de unicidade das URLs de um servidor. A verificação de unicidade de URLs no sistema VEUNI é realizada por meio de uma série de intercalações das URLs que já estão armazenadas em R_i em disco com as URLs de L_{id} e E_{id} que estão na memória. Veremos na próxima subseção que o passo de intercalação é realizado sobre cada entrada S_{ij} e as URLs do Servidor_{ij} encontradas no ciclo de coleta.

3.2.3 Intercalação de S_{ij} com B_j

Esta subseção descreve o passo de intercalação de S_{ij} com B_j, apresentado na linha 6 do Programa 2. O passo de intercalação é responsável por garantir a unicidade das

URLs da coleta. As novas URLs identificadas no ciclo de coleta são intercaladas com as URLs armazenadas em disco. Simultaneamente, os metadados referentes às URLs que foram coletadas no ciclo de coleta são atualizados. Na linha 5, encontramos o laço que percorre cada servidor S_{ij} para que dois passos sejam realizados:

- Inserir em R_i novas URLs de servidores que já existem no repositório.
- Reescrever em R_i as URLs de servidores que não sofreram modificação.

Programa 3 Refinamento do algoritmo de intercalação de S_{ij} com B_i

Entrada: S_{ij}: URLs do Servidor$_{ij}$ existentes em R_i;
B_j: URLs coletadas, metadados e URLs extraídas do Servidor$_{ij}$.
Saída: Snovo$_{ij}$: resultado da intercalação de B_{ij} com S_{ij}.
1: Carrega URLs$_{ij}$ para a memória utilizando as informações de S_{ij}.
2: **se** B_j != vazio **então**
3: Intercala o conjunto URLs$_{ij}$ com o conjunto de URLs de B_j.
4: Armazena em Snovo$_{ij}$ o resultado da intercalação realizada.
5: **se não**
6: Armazena S_{ij} em Snovo$_{ij}$.

No Programa 3 é apresentado o algoritmo utilizado para realizar o passo de intercalação. Na linha 1 do Programa 3, as URLs do Servidor$_{ij}$ armazenadas em URLs$_{ij}$ são carregadas para a memória. O próximo passo consiste em verificar se há necessidade de realizar uma intercalação entre o conjunto URLs$_{ij}$ e o conjunto B_{ij}, que contém as URLs do Servidor$_{ij}$ que foram encontradas no ciclo de coleta. Se o conjunto B_{ij} for vazio, então não houve atualização das URLs desse servidor e também não houve extração de URLs desse servidor, portanto não há necessidade de realizar a intercalação e o Snovo$_{ij}$ simplesmente recebe o valor atual de S_{ij} (linha 6).

Caso o conjunto B_j seja diferente de vazio, as URLs de B_j devem ser incorporadas ao conjunto URLs$_{ij}$. Na linha 3 do Programa 3 é realizada a intercalação entre o conjunto URLs$_{ij}$ e o conjunto de URLs de B_j. O algoritmo utilizado para ilustrar os procedimentos executados para intercalar esses dois conjuntos é apresentado no Programa 4. Na linha 4 do Programa 3, o resultado da intercalação é armazenado em Snovo$_{ij}$.

A intercalação das URLs de um Servidor$_{ij}$ deve ser realizada em dois passos: (i) intercala as URLs da categoria *pré-escalonadas* do conjunto URLs$_{ij}$ com as URLs do conjunto B_j coletadas no ciclo de coleta; (ii) intercala as URLs da categoria *restantes* do conjunto URLs$_{ij}$ com as URLs do conjunto B_j extraídas no ciclo de coleta.

Programa 4 Algoritmo para intercalar as URLs do Servidor$_{ij}$ já armazenadas em R_i (URLs$_{ij}$) com as URLs desse servidor encontradas no ciclo de coleta (B_j).

Entrada: URLs$_{ij}$: URLs do Servidor$_{ij}$ que estão armazenadas em R_i;
B_j: URLs do Servidor$_{ij}$ encontradas no ciclo de coleta.
Saída: URLsnovo$_{ij}$: resultado da intercalação de URLs$_{ij}$ com B_j.

1: Separa as URLs do conjunto URLs$_{ij}$ em URLs_PreEscalonadas$_{ij}$ e URLs_Restantes$_{ij}$.
2: Separa as URLs do conjunto B_j em B_Coletadas$_j$ e B_Extraidas$_j$.
3: Ordena lexicograficamente os conjuntos B_Coletadas$_j$ e B_Extraidas$_j$.
4: Intercala as URLs dos conjuntos URLs_PreEscalonadas$_{ij}$ e B_Coletadas$_j$ e armazena em um *buffer* chamado *pre_escalonadas*.
5: Intercala as URLs dos conjuntos URLs_Restantes$_{ij}$ e B_Extraidas$_j$ e armazena em um *buffer* chamado *restantes*.
6: Redistribui as URLs dos *buffers pre_escalonadas* e *restantes* se for necessário.
7: Armazena em URLsnovo$_{ij}$ as URLs dos conjuntos *pre_escalonadas* e *restantes*.

Na linha 1 do Programa 4 as URLs do conjunto URLs$_{ij}$ são separadas em URLs_PreEscalonadas$_{ij}$ e URLs_Restantes$_{ij}$. Na linha 2, ocorre a separação das URLs de B_j nos conjuntos B_Coletadas$_j$ e B_Extraidas$_j$. A premissa para fazer corretamente a intercalação entre dois conjuntos consiste de eles estarem previamente ordenados. As URLs dos conjuntos URLs_PreEscalonadas$_{ij}$ e URLs_Restantes$_{ij}$ já estão ordenadas lexicograficamente, pois elas são armazenadas em URLs$_{ij}$ dessa forma. Então só resta ordenar as URLs dos conjuntos B_Coletadas$_j$ e B_Extraidas$_j$. As duas ordenações são realizadas na linha 3 do Programa 4.

Após os quatro conjuntos de URLs estarem ordenados lexicograficamente, as intercalações são realizadas nas linhas 4 e 5. A intercalação é realizada atualizando os metadados das URLs coletadas, adicionando novas URLs encontradas e eliminando as URLs repetidas.

O próximo passo do Programa 4 é redistribuir as URLs retornadas das intercalações entre as categorias *pré-escalonadas* e *restantes*. Essa redistribuição é realizada na linha 6. Por último, na linha 7 do Programa 4, os conjuntos *pre_escalonadas* e *restantes* retornados da redistribuição são armazenados em URLsnovo$_{ij}$.

A redistribuição de URLs entre as categorias *pré-escalonadas* e *restantes* é realizada utilizando alguma política de redistribuição. Uma política de redistribuição que vise uma expansão do número de servidores distintos existentes na coleta deve selecionar para a categoria *pré-escalonadas*, URLs da categoria *restantes* que ainda não foram coletadas. Já uma política de redistribuição de URLs que priorize o refrescamento de um conjunto de páginas que frequentemente é atualizado, deve selecionar para a categoria *pré-escalonadas*, as URLs referentes a essas páginas. A política de redistribuição

que adotamos nos experimentos realizados com o sistema VEUNI foi a de selecionar para a categoria *pré-escalonadas* 1000 URLs ainda não coletadas da categoria *restantes* sempre que o número de URLs a coletar na categoria *pré-escalonadas* fosse menor que 10.

3.2.4 Inserção de Novos Servidores

Esta subseção descreve o passo de inserção de novos servidores no repositório R_i, apresentado na linha 9 do Programa 2. O algoritmo utilizado para inserir em Rnovo$_i$ as URLs armazenadas em B_j de servidores que ainda não existem em R_i é apresentado no Programa 6. Na linha 1 do Programa 6, as URLs de B_j são dividas nas categorias *pré-escalonadas* e *restantes* e o resultado da divisão é armazenado em B_PreEscalonadas$_{ij}$ e B_Restantes$_{ij}$. Na linha 2, as URLs de B_PreEscalonadas$_{ij}$ e B_Restantes$_{ij}$ são armazenadas em Rnovo$_i$.

Programa 5 Algoritmo para inserir em Rnovo$_i$ as URLs do novo servidor B_j.

Entrada: B_j: URLs coletadas, metadados e URLs extraídas agrupados por servidor.
Saída: Rnovo$_i$: Repositório atualizado com as URLs de B_{ij}.

1: Divide as URLs de B_j nas categorias *pré-escalonadas* e *restantes* e armazena o resultado em B_PreEscalonadas$_{ij}$ e B_Restantes$_{ij}$.
2: Insere B_PreEscalonadas$_{ij}$ e B_Restantes$_{ij}$ em Rnovo$_i$.

Do mesmo modo que a redistribuição das URLs entre as categorias *pré-escalonadas* e *restantes* é realizada considerando uma política de redistribuição, a divisão das URLs entre as categorias *pré-escalonadas* e *restantes* requer uma política de divisão. A política de divisão utilizada pode, por exemplo, ter o mesmo princípio da política de redistribuição. Nos experimentos realizados utilizando o sistema VEUNI, a política de divisão adotada foi selecionar as 1000 URLs com os menores caminhos. É interessante inserir as URLs com os menores caminhos primeiramente na categoria *pré-escalonadas*, visto que normalmente mais URLs distintas são encontradas nas páginas mais próximas ao diretório raiz do servidor.

3.3 Soluções Alternativas

Nesta seção, apresentamos três abordagens inicialmente consideradas para a verificação de unicidade de URLS: gerenciador de banco de dados *chave/valor*, tabela de alocação de arquivos do UNIX e repositório central. Estas três abordagens foram implementadas,

mas se mostraram menos eficiente que o sistema VEUNI descrito na Seção 3.1, conforme detalhado a seguir.

3.3.1 Gerenciador de Banco de Dados *Chave/Valor*

É natural pensar que o problema de verificação de unicidade de URLs pode ser resolvido utilizando um gerenciador de banco de dados que armazene pares (*chave/valor*), considerando as URLs como chaves primárias. Assim nossa primeira abordagem para solução do problema de verificação de unicidade de URLs foi utilizar esse tipo de gerenciador de banco de dados.

O Tokyo Cabinet[1] é uma biblioteca de rotinas para gerenciamento de banco de dados. O banco de dados gerenciado pelo Tokyo Cabinet é simplesmente um arquivo contendo registros que são pares *chave/valor*. Cada *chave* e *valor* são representados por uma sequência de bytes de tamanho variável. Além disso, o Tokyo Cabinet permite que as informações sejam armazenadas em formato binário ou como uma sequência de caracteres. Esses registros são organizados como tabela-hash, árvore B+ ou *array* de tamanho fixo. Devido a sua eficiência em espaço de armazenamento e tempo de processamento, o Tokyio Cabinet é reconhecido como o sucessor do GDBM (*GNU Database Manage*) e do QDBM (*Quick DataBase Manager*).

Ao utilizar o Tokyo Cabinet como verificador de unicidade de URLs, foi observado na prática que o gargalo do coletor se concentrou justamente nesse componente. Em particular, observamos que o motivo que tornou o verificador de unicidade de URLs o gargalo da arquitetura do coletor foi a ineficiência na realização de consultas ao banco de dados onde as URLs eram armazenadas. Cada consulta realizada era demorada e milhões de consultas eram realizadas em cada ciclo de coleta. Considerando o alto desempenho[2] relatado sobre o Tokyo Cabinet, abandonamos a utilização de um gerenciadores de banco de dados como verificador de unicidade de URLs.

3.3.2 Tabela de Alocação de Arquivos do UNIX

Esta estratégia é baseada na idéia de realizar uma separação das URLs por servidor e armazená-las em um arquivo identificado pelo nome do servidor. Dessa forma, quando necessitássemos incluir uma nova URL pertencente a um servidor já armazenado no disco, o arquivo com as URLs desse servidor deveria ser lido, a URL deveria ser incorporada ao conjunto já existente e o resultado deveria ser reescrito no disco. Entretanto, essa solução não se mostrou adequada.

[1]Disponível em http://fallabs.com/tokyocabinet/
[2]Disponível em http://tokyocabinet.sourceforge.net/benchmark.pdf

Ao realizarmos testes de desempenho dessa arquitetura, observamos que seu desempenho era substancialmente degradado na medida que o número de servidores identificados aumentava. A falha dessa abordagem pode ser explicada por dois motivos. O primeiro motivo foi a utilização do índice de alocação de arquivos do UNIX tão frequentemente para encontrar a localização do arquivo de cada servidor requisitado. O outro motivo foi a extensa fragmentação do disco obtida com as inúmeras reescritas dos arquivos de cada servidor.

3.3.3 Repositório Central

O algoritmo repositório central foi desenvolvido no intuito de minimizar a sobrecarga de utilização do índice de alocação de arquivos do UNIX. Esse algoritmo realiza a verificação de unicidade de URLs em *batch*, como o sistema VEUNI (Seção 3.1). Dessa forma, a busca individual de URLs no repositório de URLs armazenado em disco é evitada e um bloco inteiro de URLs é tratado a cada iteração. A seguir apresentamos o primeiro algoritmo especificado para tratar as URLs em *batch* (Programa 6).

Programa 6 Algoritmo para realizar a verificação de unicidade de URLs em *batch*.
Entrada: U: Conjunto de URLs encontradas no ciclo de coleta.
Saída: Repositório R de URLs existentes no coletor atualizado com as URLs de U.
1: Rnovo recebe a intercalação de R com U.
2: R recebe Rnovo.

A entrada do Programa 6 é o conjunto U de URLs encontradas no ciclo de coleta e a saída é o repositório R de URLs existentes no coletor atualizado com as URLs do ciclo de coleta corrente. A intercalação do conjunto de URLs U com o repositório R, linha 1 do Programa 6, é realizada respeitando os seguintes princípios: (i) as URLs de U que já existem em R são descartadas; (ii) as URLs de U que não existem em R são inseridas em R; (iii) as URLs de R que não existem em U são preservadas. O problema dessa abordagem é que o repositório R precisaria ser carregado para a memória para a realização da intercalação. Entretanto, como o repositório R armazena todas as URLs que o coletor conhece, ele não cabe na memória.

Uma solução que encontramos para esse problema foi dividir R em subgrupos S_i, tal que cada S_i caiba na memória, conforme ilustra a Figura 3.5. Uma forma de fazer isso é garantindo que cada S_i possui apenas URLs de um mesmo servidor i. Observe que, na prática, não ocorre de S_i não caber na memória, pois o número de URLs de um mesmo servidor não é grande o suficiente para isso.

Além de dividir R em subgrupos S_i, é necessário dividir U em subgrupos B_i, tal que B_i possua também apenas URLs do mesmo servidor i, para ser possível realizar

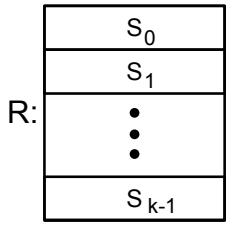

Figura 3.5. Repositório R dividido em subgrupos S_i.

as intercalações. Com a divisão de R em S_i e a divisão de U em B_i, apresentamos um novo algoritmo para verificar a unicidade de URLs (Programa 7).

Programa 7 Algoritmo para realizar a verificação de unicidade de URLs em *batch* utilizando um repositório central R dividido em subgrupos S_i.

Entrada: R: Repositório de URLs existentes no coletor dividido em subgrupos S_i;
U: Conjunto de URLs encontradas no ciclo de coleta.
Saída: Repositório R de URLs existentes no coletor atualizado com as URLs de U.

```
1: Particione U em subgrupos B_i, tal que B_i possua URLs de um mesmo servidor i.
2: para todo S_i em R faça
3:     Intercalação de S_i com B_i.
4:     Grava em Rnovo o resultado da intercalação.
5: para todo B_i de U que ainda não foi inserido em Rnovo faça
6:     Grava B_i em Rnovo.
7: R recebe Rnovo.
```

Observe que no Programa 7, o repositório R é completamente reescrito no disco sempre que um conjunto U precisa ser inserido em R (linha 7). Essa tarefa implica na reescrita de muitas URLs de R que não foram modificadas no ciclo de coleta, o que representa um desperdício de tempo. Tal reescrita de URLs não modificadas poderia ser minimizada se múltiplos repositórios R_i fossem utilizados no lugar de um único. Por esse motivo, o sistema VEUNI proposto na Seção 3.1 considera múltiplos repositórios R_i.

Capítulo 4

Resultados Experimentais

Neste capítulo descrevemos os experimentos e resultados obtidos. Na Seção 4.1 apresentamos o algoritmo *baseline* utilizado nos experimentos realizados. Na Seção 4.2 apresentamos a metodologia utilizada nos experimentos. Na Seção 4.3 discutimos os resultados obtidos experimentalmente.

4.1 Algoritmo DRUM como *Baseline*

A avaliação do desempenho do sistema VEUNI é realizada utilizando uma implementação do algoritmo DRUM (*Disk Repository with Update Management*). O DRUM foi utilizado como verificador de unicidade de URLs nas coletas realizadas pela máquina de busca IRLBot [Lee et al., 2009]. A escolha desse algoritmo como *baseline* se deve ao fato dele ser considerado o algoritmo estado-da-arte no problema de verificação de unicidade de URLs.

Descrição do Algoritmo

O objetivo do DRUM é permitir o armazenamento eficiente de grandes quantidades de pares <*key,value*>, onde *key* é um identificador único (*hash*) de algum dado e *value* são informações arbitrárias relacionadas à *key*. O DRUM suporta três operações básicas sobre os pares armazenados: *check*, *update* e *check+update*.

A operação *check* possui como entrada um conjunto contendo chaves que devem ser verificadas contra o conjunto de chaves já armazenadas na estrutura de dados *disk cache*. Durante a verificação, cada chave do conjunto de entrada é classificado como sendo única ou duplicada. Para chaves duplicadas, o valor associado a cada chave pode ser opcionalmente recuperado do disco e utilizado para algum processamento.

A operação *update* possui como entrada uma lista de pares <*key,value*> que devem ser intercalados com a estrutura de dados *disk cache* existente. Se uma dada chave já existe, seu valor é atualizado (sobrescrito ou incrementado, por exemplo). Caso contrário, uma nova entrada é criada no *disk cache*.

Já a operação *check+update* é uma união entre as duas primeiras operações. Esta operação realiza a checagem (*check*) e atualização (*update*) de um conjunto de chaves fornecido como entrada em apenas uma passada pelo *disk cache*.

Uma visão em alto nível do algoritmo DRUM é ilustrada na Figura 4.1. Observe na Figura 4.1 que tuplas <*key,value,aux*>, onde *aux* é uma informação auxiliar associada a cada chave, chegam ao algoritmo DRUM. Para tratar as tuplas <*key,value,aux*> que estão chegando é realizada uma distribuição dos pares <*key,value*> entre k *buckets* $Q_1^H ... Q_k^H$ no disco. Esta distribuição é baseada nos valores de *key*, ou seja, todas as chaves de um mesmo *bucket* possuem como prefixo a mesma sequência de *bits*.

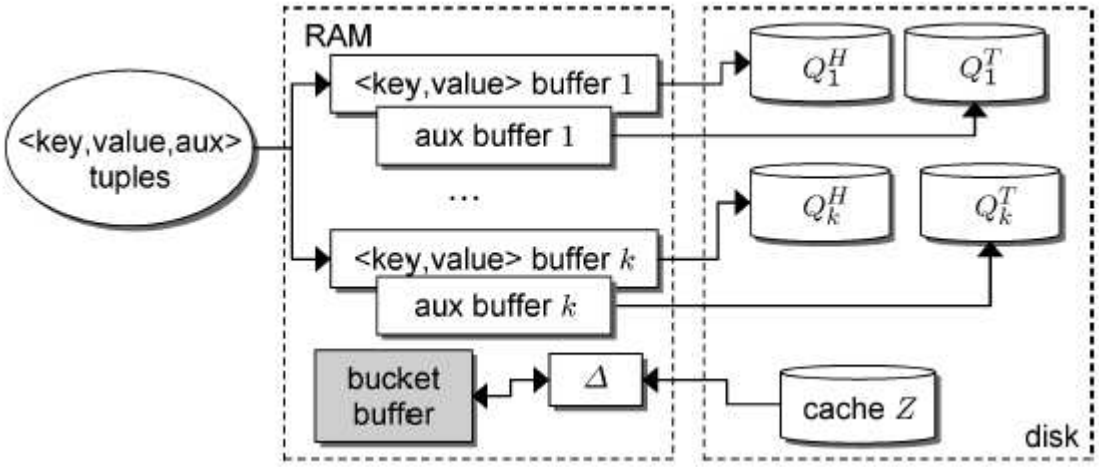

Figura 4.1. Arquitetura e funcionamento do DRUM [Lee et al., 2009].

Os pares <*key,value*> são primeiramente armazenados em k *memory arrays* de tamanho M. Quando um *memory array* enche, seu conteúdo é escrito no disco. O valor *aux* de cada chave presente no i-ésimo *bucket* é mantido em um arquivo separado chamado Q_i^T respeitando a ordem FIFO dos pares <*key,value*> armazenados em Q_i^H. O valor *aux* de cada chave pode ser utilizado para armazenar a sequência de caracteres de uma URL. Vale ressaltar que para manter a eficiência de leitura e escrita do disco e evitar segmentação na tabela de alocação de arquivos, os *buckets* são pré-alocados no disco antes de serem utilizados.

Uma vez que o *bucket* preenchido com mais informações atinge um certo tamanho $r < R$, o seguinte processo é repetido para $i = 1, ..., k$:

1. O conteúdo do bucket Q_i^H é armazenado de forma ordenada na estrutura de dados *bucket buffer* ilustrada na Figura 4.1.

2. O arquivo do disco Z é sequencialmente lido em pedaços de tamanho Δ bytes e suas chaves são comparadas com as chaves presentes no *bucket* Q_i^H para determinação de unicidade.

3. Os pares *<key,value>* em Q_i^H que requerem uma atualização são intercalados com o conteúdo do *disk cache* e armazenados em uma versão atualizada de Z.

4. Após todas as chaves únicas em Q_i^H serem encontradas, a ordem FIFO dessas chaves é restaurada, Q_i^T é sequencialmente lido e armazenado na memória em blocos de tamanho Δ e os valores *aux* referentes as chaves únicas podem ser utilizados para processamento.

Obs.:Um aspecto importante deste algoritmo é que todos os *buckets* são checados em apenas uma passagem pelo repositório Z.

Com a descrição do algoritmo DRUM realizada, é possível mostrar como o DRUM é utilizado em um coletor para armazenar as URLs e as informações relacionadas às páginas coletadas. As URLs coletadas e as URLs extraídas são carregadas para a memória principal da mesma maneira que é realizada no sistema VEUNI. Cada URL é utilizada para formar uma tupla *<URLhash,-,URLtext>*, onde *key* é um *hash* de 8 bytes da URL, *value* é vazio e *aux* é a sequência de caracteres da URL. Após a geração destas tuplas, o próximo passo consiste em começar a inserção das URLs no repositório do DRUM.

A inserção das URLs no repositório central do DRUM é realizada em duas etapas. A primeira etapa consiste em inserir as URLs que foram coletadas. Para fazer isso, é necessário invocar a operação *update*. A segunda etapa consiste em inserir as URLs que foram extraídas das páginas. Antes de realizar a inserção das URLs extraídas é necessário invocar a operação *check* para verificar quais URLs ainda não existem no repositório. Por último, a operação *update* é invocada para o conjunto de URLs que não existem no repositório. Uma alternativa mais eficiente para inserir as URLs que não existem no repositório, é utilizar a operação *check+update* sobre o conjunto de URLs extraídas.

Principais Diferenças entre o VEUNI e o DRUM

A primeira grande diferença do algoritmo DRUM em relação ao sistema VEUNI é o fato do algoritmo DRUM utilizar um repositório central para armazenar as URLs. Vale

lembrar que o sistema VEUNI divide o repositório de URLs em blocos R_i, $0 \leq i < N$. A utilização de um repositório central para armazenar as URLs é um ponto negativo do algoritmo DRUM, visto que é necessário reescrever todo repositório central no disco sempre que uma atualização no repositório é realizada. Como visto na descrição do algorimo DRUM, a atualização do repositório central é realizada quando um dos *buckets* em disco se enche. Nesse momento, o conteúdo de todos os *buckets* são intercalados com o repositório central.

Uma outra diferença entre o algoritmo DRUM e o sistema VEUNI é que o algoritmo DRUM realiza diversas reescritas do repositório de URLs em um único ciclo de coleta. Essa abordagem é diferente do que é realizado pelo sistema VEUNI, pois o VEUNI atualiza apenas uma vez um bloco R_i de URLs em um ciclo de coleta. Esse fato é uma desvantagem do algoritmo DRUM, pois como são realizadas inúmeras atualizações no repositório de URLs, o tempo necessário para realizar um ciclo de coleta se torna maior.

Implementação do Algoritmo

Com a arquitetura original do DRUM apresentada, podemos agora discutir a implementação do DRUM que utilizamos. Esta implementação está disponível publicamente na Web[1]. Trata-se de um projeto desenvolvido utilizando a linguagem C++ e algumas bibliotecas de código-aberto: *Boost* e *Berkeley DB*. A seguir descrevemos os detalhes de implementação desse algoritmo.

Os *buckets* foram pré-alocados no disco com tamanho de 150 MB. No total foram pré-alocados 8 *buckets*. É importante que os *buckets* já estejam alocados antes da execução do algoritmo, pois caso contrário uma fragmentação do disco seria inevitável, devido as inúmeras atualizações que ocorrem nesta estrutura de dados.

Os *memory arrays* foram implementados utilizando a *boost::array* e o tamanho máximo dos *arrays* foi definido como 10000 URLs. Vale lembrar que quando um *array* se enche, seu conteúdo é transferido para o seu *bucket* correspondente. O tamanho máximo do *array* é utilizado apenas para indicar o momento adequado para essa transferência ser realizada.

O repositório persistente Z foi implementado como um banco de dados *Berkeley DB* utilizando uma *BTree* para armazenamento. Esse ponto pode ser considerado uma deficiência dessa implementação. O tamanho máximo R foi definido como 100 MB. Assim quando algum *bucket* atinge esse valor limite, os conteúdos de todos *buckets*

[1] http://www.codeproject.com/KB/recipes/cppdrumimplementation.aspx

existentes são agregados ao conteúdo do repositório persistente Z. Essa agregação é realizada por meio da intercalação entre os conjuntos de chaves.

As chaves com menores prefixos são armazenadas no *bucket* 0, enquanto as chaves com maiores prefixos são armazenadas no bucket 7. Quando os *buckets* são carregados na memória para sincronização com o repositório persistente, ocorre um alto nível de localidade nas consultas submetidas ao *Berkeley DB*. Este fato implica em uma velocidade maior de acesso e em uma diminuição de chamadas I/O, pois o *Berkeley DB* oferece um cache transparente na memória para chaves frequentemente consultadas. Vale ressaltar que no intuito de obter eficiência na realização da sincronização entre os *buckets* e o repositório Z, a iteração sobre as chaves existentes nos *buckets* deve respeitar a ordem natural de iteração sobre as chaves do banco de dados do *Berkeley DB*.

4.2 Metodologia

Uma maneira de observar o desempenho do sistema que estamos propondo é realizar uma simulação de coleta. Em uma coleta simulada, a cada ciclo do coletor, teríamos um novo conjunto de URLs coletadas, um novo conjunto de metadados das páginas coletadas e um novo conjunto de URLs extraídas que são as principais entradas do sistema VEUNI.

A simulação de uma coleta é preferível a realizar de fato a coleta, pois não precisamos utilizar os outros componentes do coletor (*fetcher*, extrator de URLs e escalonador) e podemos assim focar apenas no componente que está sendo estudado neste trabalho. Além disso, o experimento se torna mais controlado, com limites bem definidos e com uma facilidade maior de reprodução. As URLs que irão compor a coleta simulada foram obtidas da coleção de páginas da Web ClueWeb09. A seguir descrevemos em detalhes como a coleção ClueWeb09 foi construída.

A Coleção ClueWeb09

A ClueWeb09 é uma coleção de 25 terabytes com aproximadamente 1 bilhão de páginas da Web, que foi gerada entre janeiro e fevereiro de 2009. A ordem utilizada para realizar a coleta foi a *best-first search*, usando a métrica OPIC (*Online Page Importance Computation*). A coleta se iniciou de uma semente de aproximadamente 29 milhões de URLs.

Na métrica OPIC, o valor atribuído a uma página é igualmente distribuído entre as páginas que ela aponta. Essa métrica é similar ao *PageRank* [Page et al., 1998], mas é mais rápida e o seu cálculo é realizado em apenas uma etapa [Abiteboul et al., 2003].

Na construção da semente, 20 milhões de URLs foram obtidas selecionando as URLs com maiores valores OPIC analisando uma coleta de 200 milhões de páginas da língua inglesa feita durante janeiro e junho de 2008 [Abiteboul et al., 2003]. As 9 milhões de URLs restantes foram as URLs que apareceram no topo dos resultados de máquinas de buscas comerciais para um conjunto de 4 mil consultas. A seleção das consultas a serem submetidas às máquinas de busca foi feita utilizando diversas abordagens.

A primeira abordagem utilizada para obter consultas foi utilizando o *log* da AOL. Primeiramente, as 1050 consultas mais frequentes do *log* de consultas da AOL foram selecionadas. Por último, mais 1050 consultas foram selecionadas aleatoriamente do restante do *log* de consultas da AOL.

Outra alternativa para geração de consultas foi utilizar os nomes de categorias do DMOZ[2]. No total foram criadas 2000 consultas utilizando essas categorias. O critério utilizado na seleção foi o tamanho das categorias. As 2000 maiores categorias foram selecionadas. As consultas criados utilizando o DMOZ foram interessantes por serem consultas mais genéricas.

A maioria das consultas obtidas até então estavam escritas na língua inglesa e o esperado era que essas consultas iriam produzir como resultado URLs da língua inglesa. Por esse motivo, os pesquisadores também utilizaram uma estratégia de criação de consultas baseada na tradução das consultas da língua inglesa. As consultas foram traduzidas para português, chinês, espanhol, japonês, alemão, francês, arábico, italiano e coreano.

Após a definição do conjunto de consultas, cada uma delas foi submetida a duas máquinas de buscas comerciais. As top 500 URLs por consulta retornadas pelo Google e Yahoo! foram utilizadas na construção da semente. Os resultado de máquinas de buscas comerciais foram utilizados, pois é esperado que a maioria das páginas retornadas possuam *PageRank* elevado.

As páginas coletas estão armazenadas em arquivos no formato WARC. Cada arquivo WARC possui aproximadamente o tamanho descomprimido de 1 gigabyte e contém aproximadamente 40 mil páginas. Na Tabela 4.1 pode ser visualizado a distribuição de páginas da ClueWeb09 por idioma.

A coleção ClueWeb09 é distribuída pela *Carnegie Mellon University* apenas para fins de pesquisa. Ela pode ser obtida assinando um contrato com a *Carnegie Mellon University* e pagando uma taxa que cobre o custo de manutenção e distribuição da coleção. Essa coleção foi essencial para a realização dos experimentos deste trabalho,

[2]O *Directory Mozilla* (DMOZ), também conhecido como *Open Directory Projet*(ODP), é um projeto que conta com a colaboração de voluntários que editam e categorizam páginas da internet.

Inglês:	503.903.810	Alemão:	49.814.309
Chinês:	177.489.357	Português:	37.578.858
Espanhol:	79.333.950	Arábico:	29.192.662
Japonês:	67.337.717	Italiano:	27.250.729
Francês:	50.883.172	Coreano:	18.075.141

Tabela 4.1. Distribuição de páginas da ClueWeb09 por idioma.

pois com ela foi possível simular uma coleta para avaliarmos o desempenho do sistema VEUNI. A simulação da coleta foi realizada extraindo da coleção 350 milhões de URLs. Com esse conjunto de URLs, foi possível simular uma coleta experimentando diversos tamanhos de escalonamento e outros parâmetros internos dos algoritmos avaliados.

4.2.1 Experimento 1: Requisitos de Tempo e de Espaço

No primeiro experimento iremos simular uma coleta que contenha 350 milhões de URLs. A coleta simulada será realizada em 350 ciclos de coleta, onde cada ciclo conterá 1 milhão de URLs. O conjunto de 1 milhão de URLs será dividido em dois subconjuntos: URLs coletadas (100 mil URLs) e URLs extraídas (900 mil URLs). Na simulação da coleta, iremos mensurar duas grandezas importantes para os algoritmos de verificação de unicidade de URLs:

- Tempo necessário para atualização das estruturas de dados em cada ciclo de coleta.

- Espaço em disco necessário para armazenar as estruturas de dados em cada ciclo de coleta.

Além de medir o desempenho do sistema VEUNI, o desempenho do algoritmo definido como *baseline* será medido. Como visto na Seção 4.1, o algoritmo que iremos utilizar como *baseline* é uma implementação do algoritmo DRUM. A comparação dos dois algoritmo será realizada acompanhando o tempo de processamento requerido pelo sistema VEUNI e pelo algoritmo *baseline* na tarefa de armazenar 350 milhões de URLs de uma coleta simulada.

Com relação ao requisito de espaço, o algoritmo DRUM e o sistema VEUNI demandam aproximadamente a mesma quantidade de espaço para as suas execuções.

4.2.2 Experimento 2: Taxa de Coleta

O segundo experimento que iremos realizar possui como objetivo mostrar o tempo necessário para realizar uma coleta quando é utilizado o sistema VEUNI como verificador de unicidade de URLs e quando é utilizado o algoritmo *baseline*. Para isso é necessário saber qual o tempo médio gasto em cada ciclo de coleta nos outros componente do coletor (fetcher, extrator de URLs e escalonador).

No experimento iremos simular uma coleta de 35 milhões de URLs, realizando 350 ciclos de coleta. Em cada ciclo de coleta, 100 mil URLs são coletadas e 900 mil URLs são extraídas. Portanto, ao terminar a coleta simulada, o coletor terá coletado 35 milhões de URLs e irá conhecer 350 milhões de URLs.

Este experimento é possível de ser simulado, pois nos testes do sistema VEUNI realizamos uma coleta utilizando toda arquitetura do coletor InWeb e armazenamos os tempos de processamento requerido por cada componente. Essa coleta recebeu o nome de WBR2010. As estatísticas sobre a coleta WBR2010 podem ser visualizadas na Tabela 4.2.

URLs coletadas	189,355,270
URLs estáticas	62,465,536
URLs dinâmicas	88,518,835
URLs a coletar	188,501,674
URLs conhecidas	377,856,944
Período de coleta	22/09/2010 a 05/10/2010
Tempo efetivo de coleta	∼7 dias

Tabela 4.2. Estatísticas da coleta WBR2010.

O tempo médio gasto pelos componentes *fetcher*, extrator de URLs e escalonador para coletar 100 mil páginas em um ciclo de coleta foi de 270 segundos. Esse tempo será utilizado no segundo experimento da seguinte forma: o tempo necessário para o coletor realizar um ciclo de coleta será igual ao tempo gasto pelo verificador de unicidade de URLs somado ao tempo médio gasto pelos outros componentes do coletor.

4.3 Análise dos Resultados

Nesta seção apresentamos os resultados dos experimentos realizados. Primeiramente, mostramos os resultados do experimento de inserção de 350 milhões de URLs no sistema VEUNI e no algoritmo *baseline*. Por último, apresentamos os resultados do experimento que compara o tempo necessário para realizar uma coleta com um coletor que utiliza o

sistema VEUNI com o tempo necessário para realizar uma coleta com um coletor que utiliza o algoritmo *baseline*.

O tempo necessário para atualização das estruturas de dados do sistema VEUNI e do *baseline* na tarefa de armazenar 350 milhões de URLs pode ser visualizado na Tabela 4.3. Por exemplo, na linha 3 da Tabela 4.3 podemos ver que com 100 milhões de URLs armazenadas, o *baseline* necessita de 81.24 segundos para inserir as URLs do ciclo de coleta corrente e o sistema VEUNI necessita de 3.53 segundos para realizar a mesma tarefa. Na Figura 4.2 ilustramos os tempos gastos em cada ciclo de coleta.

# de URLs (milhões)	Tempo (s)	
	Algoritmos	
	DRUM	VEUNI
1	22.17	1.35
50	53.53	2.60
100	81.24	3.53
150	114.55	4.47
200	149.11	6.11
250	196.60	6.17
300	238.53	7.26
350	284.92	9.07

Tabela 4.3. Resumo dos tempo necessários para atualização das estruturas de dados do sistema VEUNI e do *baseline* na tarefa de armazenar 350 milhões de URLs.

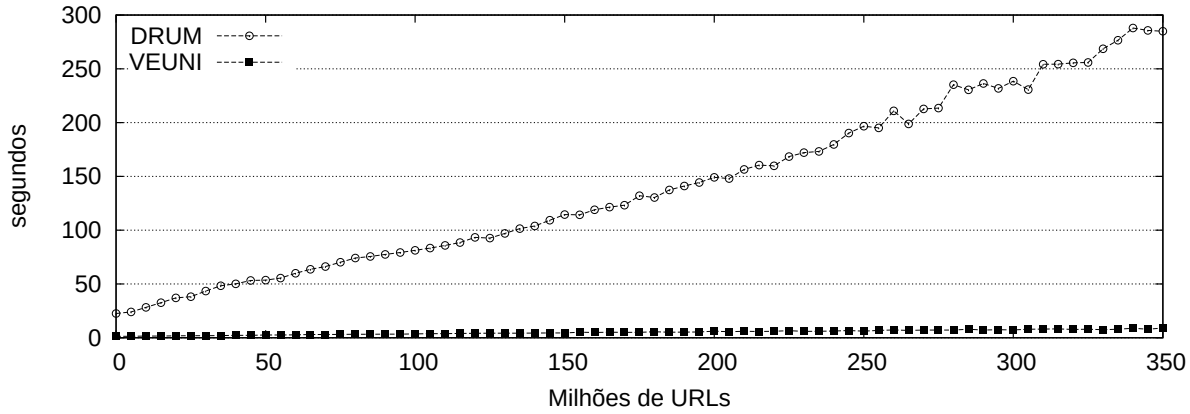

Figura 4.2. Tempo necessário para atualização das estruturas de dados do sistema VEUNI e do *baseline* na tarefa de armazenar 350 milhões de URLs.

Observe na Figura 4.2 que o tempo em geral gasto pelo sistema VEUNI foi muito inferior ao tempo gasto pelo *baseline*. Além disso, diferentemente do *baseline*, a curva

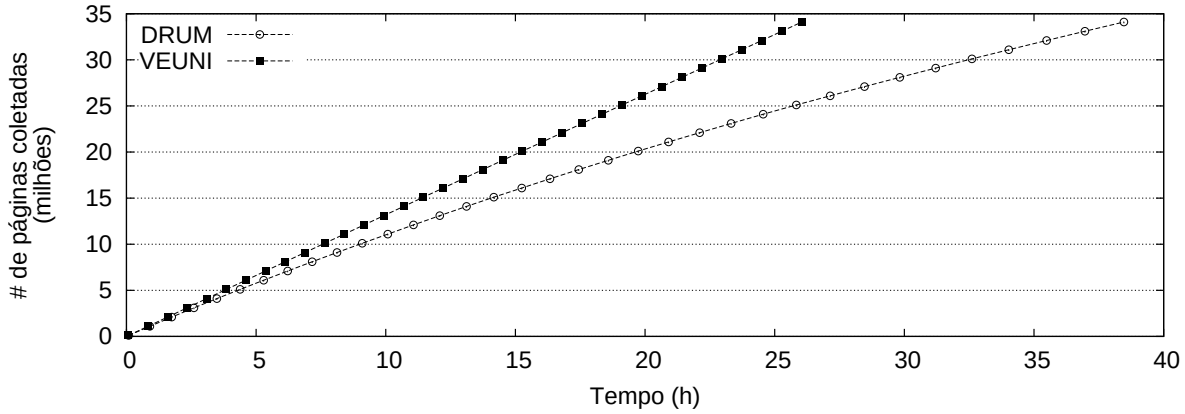

Figura 4.3. Tempo necessário para realização de uma coleta de 35 milhões de páginas utilizando o sistema VEUNI e o algoritmo *baseline*.

do sistema VEUNI praticamente se manteve constante. É possível notar que na medida que o número de URLs armazenadas no repositório aumenta, o custo de atualização dos repositórios do *baseline* também aumenta de forma significativa. Com 350 milhões de URLs inseridas, o sistema VEUNI necessita de 9.07 segundos para realizar a inserção de 1 milhão de novas URLs (Tabela 4.3). Já o *baseline* necessita de 284.92 segundos. Portanto, utilizando o algoritmo *baseline*, em cada ciclo teríamos um desperdício de aproximadamente 275 segundos apenas verificando a unicidade das URLs do ciclo corrente.

O segundo experimento que realizamos teve como objetivo simular o tempo necessário para realizar a coleta quando se utiliza o sistema VEUNI como verificador de unicidade de URLs e quando se utiliza o algoritmo *baseline*. Os resultado obtidos nesse experimento podem ser visualizados na Figura 4.3. Observe que o número de páginas coletadas por unidade de tempo pelo coletor que utiliza o algoritmo *baseline* diminui consideravelmente no decorrer da coleta, diferentemente do coletor que utiliza o sistema VEUNI. Na Tabela 4.4 podemos visualizar um resumo dos dados experimentais obtidos na simulação da coleta.

Observe na Tabela 4.4 que o coletor que utiliza o sistema VEUNI necessita de aproximadamente 26 horas para finalizar a coleta das 35 milhões de páginas. Já o coletor que utiliza o *baseline* necessita aproximadamente de 40 horas para coletar o mesmo conjunto de páginas. Com esses dados gerados, podemos calcular a taxa média de coleta obtida com os dois algoritmos. As taxas de coleta obtidas pelo coletor que utiliza o sistema VEUNI e pelo coletor que utiliza o algoritmo *baseline* são apresentadas na Tabela 4.5.

É importante observar na Tabela 4.5 como a taxa de coleta do coletor que utiliza

# de páginas coletadas	Tempo(h)	
	Algoritmos	
(milhões)	DRUM	VEUNI
0.1	0.08	0.07
5	4.29	3.77
10	8.99	7.56
15	14.06	11.37
20	19.61	15.19
25	25.70	19.03
30	32.46	22.88
35	39.84	26.74

Tabela 4.4. Tempo para realização da coleta simulada.

# de páginas coletadas	páginas/s	
	Algoritmos	
(milhões)	DRUM	VEUNI
0.1	341.85	368.49
5	323.55	367.87
10	308.84	367.03
15	296.32	366.26
20	283.19	365.57
25	270.19	364.87
30	256.68	364.18
35	244.00	363.51

Tabela 4.5. Taxa média de coleta obtida pelo coletor que utiliza o sistema VEUNI e pelo coletor que utiliza o algoritmo *baseline*.

algoritmo *baseline* diminui no decorrer da coleta. Para as primeiras 100 mil URLs coletadas a taxa de coleta obtida utilizando o sistema VEUNI foi de 368.49 páginas/s e utilizando o *baseline* foi de 341.85 páginas/s. No final da coleta, a taxa de coleta do coletor que utiliza o sistema VEUNI foi de 363.51 páginas/s e a taxa do coletor que utiliza o *baseline* foi de 244.00 páginas/s.

Observe que a diminuição da taxa obtida pelo coletor que utiliza o sistema VEUNI foi mais suave que a diminuição da taxa do coletor que utiliza o *baseline*. A taxa diminui de 368.49 para 363.51 (1,34%) no primeiro caso e diminuiu de 341.85 para 244.00 (28,62%) no segundo caso. Essa diminuição da taxa de coleta enfrentada pelo coletor

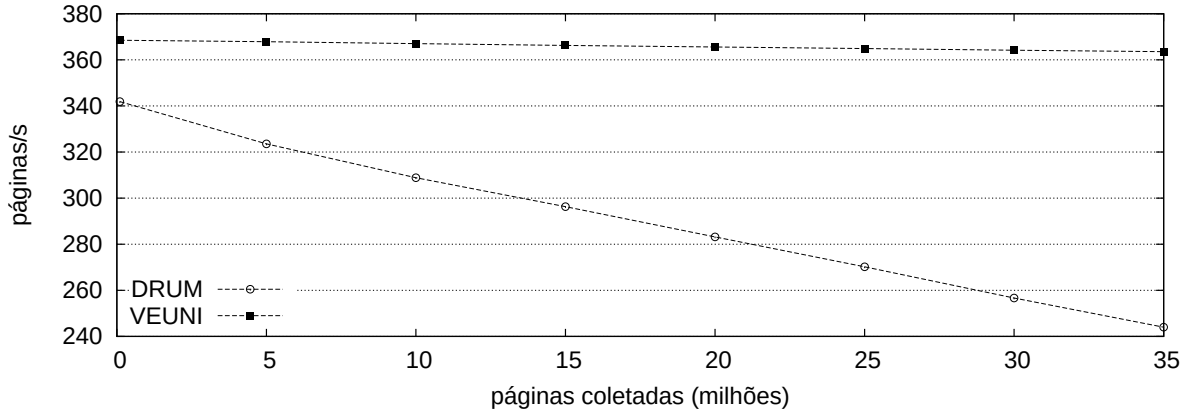

Figura 4.4. Taxa média de coleta duranta a simulação.

que utiliza o *baseline* pode ser visualizada na Figura 4.4. Nessa figura também pode ser observado que a taxa de coleta do coletor que utiliza o sistema VEUNI praticamente se mantém constante.

Com os experimentos realizados podemos chegar à conclusão de que o sistema VEUNI é superior ao algoritmo tomado como *baseline* neste trabalho, visto que, mesmo após coletar 35 milhões de páginas, ele permite que seja praticamente mantido a taxa de coleta inicial do coletor.

Capítulo 5

Conclusões e Trabalhos Futuros

Neste trabalho foi estudado o problema de verificação de unicidade de URLs em coletores de páginas Web. Esse problema é importante no contexto dos coletores de máquinas de buscas, pois caso não seja utilizado um verificador de unicidade de URLs eficaz e eficiente, o coletor de páginas web sofrerá de diversos problemas, tais como: (i) coleta indefinida de páginas já coletadas; (ii) dificuldade de expansão da coleta, visto que os mesmos servidores são sempre visitados. (iii) desperdício de recursos, como por exemplo, banda de internet e espaço de armazenamento. (iv) redução do número de páginas distintas coletadas por unidade de tempo.

A idéia principal do sistema VEUNI que propusemos para solucionar o problema de verificação de unicidade de URLs é realizar uma intercalação do conjunto de URLs identificadas em cada ciclo de coleta com o repositório que armazena as URL conhecidas pelo coletor. Uma propriedade importante encontrada no sistema VEUNI é que o repositório R de URLs é dividido em múltiplos repositórios R_i para que seja evitado a reescrita de todo conjunto de URLs conhecidas pelo coletor em cada ciclo de coleta, após a intercalação de novas URLs. Deste modo, apenas o repositório R_i utilizado pelo escalonador de URLs para gerar a lista de URLs fornecida como entrada do *fetcher*, é reescrito atualizado com as URLs coletadas no ciclo de coleta e as URLs extraídas das páginas coletadas.

Os resultados experimentais obtidos mostram que o sistema VEUNI é eficiente na tarefa de verificar a unicidade de URLs em coletores de páginas web. Seu desempenho em tempo de processamento foi superior ao algoritmo estado-da-arte (DRUM). A ineficiência do algoritmo tomado como *baseline* se deve ao fato dele realizar diversas atualizações de todo repositório de URLs em um mesmo ciclo de coleta e também ao fato dele utilizar um banco de dados *Berkeley DB* como repositório central das URLs.

Como trabalho futuro seria interessante implementar o algoritmo DRUM sem

utilizar um banco de dados *chave/valor* como repositório. Mesmo com uma implementação do algoritmo DRUM sem utilização de um gerenciador de banco de dados, é provável que o desempenho do sistema VEUNI seja superior, visto que, diferentemente do sistema VEUNI, o algoritmo DRUM ainda assim irá utilizar um repositório central para armazenar as URLs conhecidas pelo coletor. Um outro trabalho futuro interessante é realizar um estudo sobre os parâmetros utilizados no sistema VEUNI, tais como, número de blocos de URLs, número máximo de servidores armazenados em cada bloco de URLs, número máximo de URLs da categoria *pré-escalonadas*, dentre outros. O objetivo desse estudo seria encontrar os valores mais indicados para obter o máximo de eficiência na utilização do sistema VEUNI.

Referências Bibliográficas

Abiteboul, S.; Preda, M. & Cobena, G. (2003). Adaptive on-line page importance computation. In *Proceedings of the International Conference on World Wide Web*, pp. 280--290, New York, USA.

Boldi, P.; Codenotti, B.; Santini, M. & Vigna, S. (2004). Ubicrawler: a scalable fully distributed web crawler. *Software: Practice & Experience*, 34(8):711--726.

Broder, A. (2002). A taxonomy of web search. *SIGIR Forum*, 36(2):3--10.

Coffman, E. G.; Liu, Z. & Weber, R. R. (1997). Optimal robot scheduling for web search engines. *Journal of Scheduling*, 1(1):15--29.

Heydon, A. & Najork, M. (1999). Mercator: A scalable, extensible web crawler. *World Wide Web*, 2(4):219--229.

Lee, H.-T.; Leonard, D.; Wang, X. & Loguinov, D. (2009). Irlbot: Scaling to 6 billion pages and beyond. *ACM Transactions on the Web*, 3(3):1--34.

Najork, M. & Heydon, A. (2001). High-performance web crawling. Technical report, SRC Research Report 173, Compaq Systems Research, Palo Alto, CA.

Page, L.; Brin, S.; Motwani, R. & Winograd, T. (1998). The pagerank citation ranking: Bringing order to the web. Technical report, Stanford Digital Library Technologies Project, Palo Alto, CA.

Pinkerton, B. (1994). Finding what people want: Experiences with the web crawler. In *Proceedings of the International Conference on World Wide Web*, pp. 30--40, Geneva, Switzerland.

Shkapenyuk, V. & Suel, T. (2002). Design and implementation of a high-performance distributed web crawler. In *Proceedings of the International Conference on Data Engineering*, pp. 357--368, San Jose, CA.