Arquitetura e implementação de um sistema distribuído de recuperação de informação

Luiz Daniel Creão Augusto

Dissertação apresentada
ao
Instituto de Matemática e Estatística
da
Universidade de São Paulo
para
obtenção do título
de
Mestre em Ciências

Programa: Mestrado em Ciência da Computação

Orientador: Prof. Dr. Alair Pereira do Lago

Durante o desenvolvimento deste trabalho o autor recebeu auxílio financeiro do CNPq

São Paulo, fevereiro de 2010

Arquitetura e implementação de um sistema distribuído de recuperação de informação

Este exemplar corresponde à redação
final da dissertação devidamente corrigida
e defendida por (Luiz Daniel Creão Augusto)
e aprovada pela Comissão Julgadora.

Banca Examinadora:

- Prof. Dr. Alair Pereira do Lago (orientador) - IME-USP.
- Prof. Dr. Francisco Carlos da Rocha Reverbel - IME-USP.
- Prof. Dr. Sérgio Roberto Pereira da Silva - UEM.

Agradecimentos

A minha mãe e irmãs, Isabel e Raquel, pessoas fundamentais ao longo destes anos todos, por toda ajuda, carinho e suporte, sempre presentes nos momentos mais difíceis.

Ao professor, orientador, amigo e paizão Alair, por todos os conselhos, conversas e tempo investido comigo nesses últimos anos em São Paulo.

Aos amigos, por todo apoio e carinho recebido (em ordem alfabética para não despertar ciúmes!): Bárbara, Briza, Cláudio, David, Dutra, Igor, Juliana, Marinho, Maurício, Roberto, Rodrigo e Priscila.

A upLexis Tecnologia pela liberdade de utilização de sua infraestrutura e servidores no desenvolvimento desse trabalho. E também a todos os companheiros de empresa (muitos para nomear!).

E ao CNPq pelo auxílio financeiro durante o mestrado.

Resumo

A busca por documentos relevantes ao usuário é um problema que se torna mais custoso conforme as bases de conhecimento crescem em seu ritmo acelerado. Este problema passou a resolvido por sistemas distribuídos, devido a sua escalabilidade e tolerância a falhas. O desenvolvimento de sistemas voltados a estas enormes bases de conhecimento – e a maior de todas, a Internet – é uma indústria que movimenta bilhões de dólares por ano no mundo inteiro e criou gigantes.

Neste trabalho, são apresentadas e discutidas estruturas de dados e arquiteturas distribuídas que tratem o problema de indexar e buscar grandes coleções de documentos em sistemas distribuídos, alcançando grande desempenho e escalabilidade. Serão também discutidos alguns dos grandes sistemas de busca da atualidade, como o Google e o Apache Solr, além do planejamento de uma grande aplicação com protótipo em desenvolvimento.

Um projeto próprio de sistema de busca distribuído foi implementado, baseado no Lucene, com idéias coletadas noutros trabalhos e outras novas. Em nossos experimentos, o sistema distribuído desenvolvido neste trabalho superou o Apache Solr com um vazão 37,4% superior e mostrou números muito superiores a soluções não-distribuídas em hardware de custo muito superior ao nosso *cluster*.
Palavras-chave: recuperação de informação, sistemas distribuídos, arquivo invertido

Abstract

The search for relevant documents for the final user is a problem that becomes more expensive as the databases grown faster. The solution was brought by distributed systems, because of its scalability and fail tolerance. The development of systems focused on enormous databases – including the World Wide Web – is an industry that involves billions of dollars in the world and had created giants.

In this work, will be presented and discussed data structures and distributed architectures related to the indexes and searching in great document collections in distributed systems, reaching high performance and scalability. We will also discuss some of the biggest search engines, such as Google e Apache Solr, and the planning of an application with a developing prototype.

At last, a new project of a distributed searching system will be presented and implemented, based on Lucene, with ideas from other works and new ideas of our own. On our tests, the system developed in this work had throughput 37.4% higher than Apache Solr and revealed higher performance than non-distributed solutions in a hardware more expensive than our cluster.

Keywords: information retrieval, distributed systems, inverted file.

Sumário

Lista de Figuras

Lista de Tabelas

Capítulo 1

Introdução

1.1 Considerações Preliminares

Se as ferramentas de busca seqüencial (como o buscador do Windows XP e utilitário `grep` e suas variantes) parecem funcionar satisfatoriamente bem para pequenas coleções de documentos (até algumas centenas de megabytes), elas simplesmente são impraticáveis conforme as coleções aumentam de tamanho, face ao fato de que cada busca é realizada em tempo linear no tamanho da coleção.

Os sistemas de indexação e busca de bases de texto resolvem esse problema por serem capazes de pré-processar uma dada coleção de textos e gerarem índices para elas. Assim a busca por documentos que contenham determinados termos será feita de forma muito mais ágil que a varredura sequencial da coleção – desta vez em tempo linear no tamanho da busca.

Assim, com o aumento da digitalização de coleções de texto (tanto de produção nova quanto da digitalização de bases antigas), é necessário cada vez mais que bons sistemas de recuperação sejam desenvolvidos para localizar dados em documentos com alta velocidade. Hoje em dia, temos diversas ferramentas e sistemas de busca já bem desenvolvidos, entretanto a escalabilidade desses sistemas continua sendo um grande problema pelo acelerado crescimento das bases de textos.

O conceito de 'base de textos' também se expandiu com o tempo e muito além de documentos tradicionais, vieram as páginas *Web*. Com a explosão dos sistemas de busca na *Web* no final da década de 90 (como o Yahoo! e o Google), esses sistemas precisaram ser ainda mais velozes devido a colossal base de textos (a *Web*!) e com *crawlers* que constantemente vasculhassem toda a Internet atrás de novas páginas.

Nos anos mais recentes, o nicho se expandiu novamente com os indexadores de *desktops*, para se localizar documentos, contatos e *logs* dentro da imensidão dos discos rígidos das máquinas de usuários comuns (atualmente, já rompendo a casa do terabyte), com sistemas indexadores como o Google Desktop (para ambientes Windows), o Beagle (para ambientes Linux), o sistema de busca nativo no Windows 7 e o SpotLight no Mac OSX.

Outro nicho atual são os indexadores corporativos, rastreando documentos internos, memorandos, ofícios e todo tipo de 'papelada digital' de uma determinada empresa, considerando os diferentes níveis de acesso aos funcionários e que os documentos estejam distribuídos em diferentes servidores de arquivos e estações de trabalho.

Diversas formas de estruturas de dados podem ser usadas no sistema de indexação. A estrutura mais compacta e mais difundida hoje é o arquivo invertido, que necessita de espaço linear no tamanho da coleção para armazenar o índice – embora ainda seja inferior a ela, cerca 50% do tamanho. Apesar do bom funcionamento, ainda que muito menores que outras estruturas de indexação, os arquivos invertidos tem desempenho prejudicado quando crescem demais, como a maioria das estruturas de dados. Uma coleção de documentos com 400 gigabytes (um indexador de parte da *Web*, por exemplo) teria um índice invertido de cerca de 150 gigabytes.

O tempo de resposta para encontrar termos em um índice desse tamanho é seriamente comprometido, revelando um problema de escalabilidade de diversos níveis, em especial na velocidade de acesso a disco e na quantidade de memória RAM disponível.

Quando o índice passa a ser algumas vezes maior do que a memória RAM disponível, o sistema operacional precisará acionar muitas vezes o mecanismo de paginação (uma operação de entrada/saída) para carregamento das listas invertidas. Além disso, o próprio disco rígido pode ser insuficiente para armazenar a coleção e índice juntos, ou o processador pode ser incapaz de tratar a quantidade de requisições e indexações exigidas eficientemente.

Dessa forma, naturalmente a tendência desses sistemas foi a migração para ambientes distribuídos, onde múltiplos processadores, memórias e discos seriam capazes de dividir o problema para obter-se melhores resultados. Mas, como em qualquer sistema distribuído, existe uma série de problemas a serem tratados, como, por exemplo, o tipo de sistema distribuído, a comunicação entre os nós do sistema e as estruturas de dados a serem utilizadas. Além disso, deve haver um planejamento desses fatores para que o sistema continue a ser

escalável conforme o tamanho do problema cresça ainda mais.

1.2 Objetivos

Os objetivos deste trabalho são:

- Realização de estudos sobre arquiteturas distribuídas e sistemas de recuperação de informação;
- Realização de estudos sobre estruturas de dados distribuídas para recuperação de informação;
- Implementação um sistema de busca distribuído;
- A utilização deste sistema de busca distribuído para indexação de Diários Oficiais da União, um domínio carente de máquinas de busca.

1.3 Contribuições

As principais contribuições deste trabalho estão discriminadas abaixo:

- Desenvolvimento de um *patch* para o *Lucene* – uma classe que é capaz de realizar buscas de forma mais eficiente que o `ParallelSearcher`, a qual será submetida à comunidade na forma de um *patch*;
- Tablets – um estudo e implementação de um sistema de alta performance para o armazenamento e recuperação dos textos originais da coleção, um setor de pouca atenção na literatura;
- Segmentação de Índices – uma abordagem de indexação que, ao custo de um desempenho ligeiramente inferior, facilitará aspectos de balanceamento de carga e tolerância a falhas para sistemas de busca distribuídos;
- Desenvolvimento do Buscador DO – uma eficiente ferramenta de busca em Diários Oficiais de diversos estados do Brasil, superior àquela disponível no *site* da Imprensa Oficial de São Paulo.

1.4 Organização do Trabalho

No capítulo 2, apresentamos alguns trabalhos relacionados a este, com seus objetivos e sucessos. A seguir, no capítulo 3 são apresentados conceitos básicos de sistemas de buscas,

bem como as formas de distribuição de índices. No capítulo 4 são analisados os sistemas do Google e do Apache Lucene, por se tratarem da base da arquitetura deste trabalho, a ser apresentada a seguir no capítulo 5. Os experimentos com o sistema desenvolvido são demonstrados no capítulo 6. Finalmente, no capítulo 7 discutimos algumas conclusões obtidas neste trabalho.

Em anexos constam assuntos que tangeiam este trabalho como compressão de dados (apêndice B) e estratégias de *caching* para sistemas de busca (apêndice C).

Capítulo 2

Trabalhos Relacionados

Considerando a escala e o contínuo crescimento do tamanho das coleções de documentos a serem indexadas, supercomputadores raramente são uma boa escolha de hardware para sistemas de recuperação de informação: quando o desempenho do sistema começa a degradar, a opção é melhorar o hardware – uma opção caríssima e que (geralmente) é substitutiva – trocar componentes atuais por outros melhores.

Dessa forma, os sistemas distribuídos para sistemas de recuperação de informação são uma solução recorrente: quando a escala da coleção vai crescendo, espera-se apenas adicionar mais nós ao sistema. Entretanto, embora sistemas distribuídos sejam uma solução recorrente, eles estão longe de serem uma solução simples, uma vez que sua utilização exige uma série de cuidados, controles adicionais e muito esforço de desenvolvimento.

A utilização de um sistema distribuído permite a utilização de um conjunto de processadores, memórias e discos (geralmente de *commodity* hardware – máquinas comuns) que devem agir coordenadamente segundo o algoritmo distribuído implementado. Assim, a divisão das tarefas e dos dados fica a critério (e capacidade) do algoritmo desenvolvido pelo programador, que determinará (justamente com o grau de paralelismo do problema) a escalabilidade do sistema.

Os sistemas de indexação e busca possuem um alto grau de paralelismo (o que permite a criação de sistemas de porte gigantesco formado por milhares de computadores comuns, como o Google), os dados a serem distribuídos são o índice e a coleção de textos (discutidos nas seções 3 e 5.2, respectivamente), e as tarefas são a manipulação eficiente dessas duas estruturas.

Deve-se considerar também que a utilização de um sistema distribuído, e em especial

por ser constituído de máquinas comuns, acarreta em uma série de considerações que devem ser tratadas pelos algoritmos, tais como: cuidado na coordenação das tarefas, os custos de comunicação (que devem ser os menores possíveis) e as (inevitáveis) falhas de hardware que precisarão ser superadas.

Nesta seção, outros trabalhos em sistemas de recuperação de textos distribuídos serão analisados.

2.1 TodoBR

O TodoBR era uma máquina de busca da *Web* brasileira desenvolvida pela Akwan Information Technologies e pesquisadores da UFMG, adquirida pelo Google em 2005. Este projeto gerou um grande número de publicações sobre as mais diversas questões de máquinas de busca: questões de desempenho, estruturas de dados, crescimento do *corpus* indexado, dentre outros. Algumas publicações são posteriores à compra da Akwan, mas por se tratarem do mesmo grupo de pesquisadores e linha de pesquisa, continuam agrupadas aqui. Várias destas publicações estão diretamente relacionados a este trabalho.

Ribeiro-Neto et al. [30, 31] discutem a estratégia de distribuição de índices conhecida como *índices globais* (que será discutida na seção 3.2.4). Nestes trabalhos, são analisadas a validade dessas estruturas, as melhores formas de construção e aspectos de desempenho.

Badue et al. [9] compara, conceitualmente e em desempenho, as estratégias de índices globais e índices locais (que será discutida na seção 3.2.3). Alguns experimentos são realizados utilizando um pequeno *cluster* de 4 máquinas e a base de documentos TREC-3 [16]. Duas baterias de consultas são realizadas: na primeira, utilizando a bateria defina em TREC-3, as duas estratégias possuem diferença de desempenho inferior a 10%; na segunda, criando uma bateria de consultas artificiais com 2 termos (em média), os índices globais possuem melhor desempenho e a curva mostra uma tendência de melhora ainda maior conforme o *cluster* aumenta.

Em trabalho posterior, Badue [10] realiza experimentos utilizando estratégia de índices locais, um *cluster* de 8 máquinas e como base de documentos a *Web* brasileira que o TodoBR havia coletado até o ano de 2003, uma coleção 22 vezes maior que a base TREC-3. Para a bateria de consultas foi utilizado um conjunto de 10.000 consultas extraídas do *log* de consultas realizadas no TodoBR. Nos experimentos, foram analisados: o balanceamento de

carga entre os nós do *cluster*, que se mostrou muito satisfatório; a possibilidade do *broker* (a máquina que recebe as requisições e distribui entre o *cluster*) ser um ponto de gargalo do sistema, o que não acontece para índices locais; os tempos de processador e de disco para as consultas, na qual o tempo de processador se mostrou dominante, aparentemente devido ao *caching* de disco realizado pelo sistema operacional; bem como a vazão e os tempos de resposta de consultas conforme se varia a taxa de chegada de consultas por segundo ao sistema.

Em trabalho mais recente, Marin, Baeza-Yates et al. [27] novamente compara as estratégias de índices locais e índices globais, desta vez utilizando um hardware nada modesto: um *cluster* com 200 nós bi-processados e rede Infiniband. A base utilizada também é muito expressiva, com 1,5 TB de páginas coletadas da *Web* britânica. Os autores concluem que cada estratégia apresenta vantagens em diferentes cenários: para consultas booleanas do tipo *AND* (onde todos os termos devem estar presentes nos resultados), os índices locais são superiores; para consultas booleanas do tipo *OR* (onde apenas parte dos termos precisam estar presentes), os índices globais são superiores.

2.2 Google

Indiscutivelmente, o Google é o principal e o maior serviço de busca da Internet, com cerca de 14 bilhões de páginas *Web* indexadas[1]. Além de números expressivos, o Google se destaca por ter publicado uma série de trabalhos sobre sua infraestrutura interna e sistemas, em especial sobre sua estrutura inicial [13], o Map/Reduce [17], a BigTable [15] e o Google File System [20].

Como estes trabalhos se tornaram uma grande referência para a área (e, consequentemente, para este trabalho), os artigos e sistemas do Google serão abordados na seção 4.1.

2.3 Apache *Lucene*

O *Lucene* [2] é uma biblioteca *open-source* para indexação e busca em coleções de textos, desenvolvida por Doug Cutting e atualmente hospedada e mantida pela Apache Foundation. Nos últimos anos, o *Lucene* cresce em importância e utilização, possuindo uma base de desenvolvedores extremamente ativa, o que gerou livros (como [8]) e diversos subprojetos que ganharam enorme destaque, como o Solr [4] e o Hadoop [1].

O Solr é um sistema de busca baseado no *Lucene* de fácil instalação e utilização por

[1] em fevereiro/2010, segundo http://www.worldwidewebsize.com/

outros sistemas, com fácil administração via interface *Web* e consultas utilizando *web services*. Atualmente, o Solr é utilizado como solução de busca interna para dezenas *sites*[2] de *e-commerce* e conteúdo, como o *site* da Casa Branca dos EUA, a videolocadora Netflix, vários *subsites* da AOL (America On-Line) e o digg[3], uma das redes sociais de maior tráfego da Internet.

O Hadoop é uma plataforma de computação distribuída que possibilita processamento e armazenamento de petabytes de dados em até milhares de nós. O Hadoop foi inspirado pelos trabalhos publicados pelo Google citados na seção anterior, quando o *Lucene* e outros subprojetos começaram a se deparar com problemas semelhantes aos do Google. O crescimento do Hadoop é tão expressivo que se tornou base de diversos grandes serviços da Internet[4], como o Yahoo!, o Facebook, a Amazon e o Last.fm.

Por serem alternativas *open source* e utilizados neste trabalho, estes sistemas serão discutidos com mais detalhes na seção 4.2.

2.4 YaCy

Na literatura e na Internet existem algumas propostas de construção de sistemas de indexação e busca de larga escala utilizando sistemas *peer-to-peer* (P2P), tornando tais sistemas descentralizados e altamente distribuídos. Entretanto, poucos projetos desta natureza se concretizaram em sistemas reais, seja na forma de um produto comercial, seja na forma de projeto *open source*.

Se por um lado se espera que os sistemas P2P sejam altamente redundantes e de grande capacidade de armazenamento, por se basearem na estrutura da Internet para sua coordenação e divisão, por outro sofrem dos problemas crônicos decorrentes da rede: as desconexões e a latência – problemas (virtualmente) inexistentes em *clusters* e difíceis de se lidar. O problema da latência de rede deixa esses sistemas bastante distantes do ideal em termos de desempenho: as buscas são repassadas para cada um dos nós (potencialmente distantes e com diferentes latências de rede) e somente quando todos respondem que o resultado é devolvido ao usuário.

Iniciamente desenvolvido por Michael Peter Christen do Karlsruhe Institute of Technology da Alemanha, o YaCy [6] é um projeto *open source* desta natureza desenvolvido em

[2] http://wiki.apache.org/solr/PublicServers
[3] http://www.digg.com
[4] http://wiki.apache.org/hadoop/PoweredBy

Java (com versões disponíveis para Windows, Linux e Mac OS) cuja proposta da máquina de busca está na descentralização, geodistribuição e na simplicidade de adicionar novos nós na rede. Qualquer computador pode se juntar a um sistema YaCy aberto, indexando e auxiliando nas buscas da rede.

O YaCy é dividido em 4 módulos:

- Crawler – realiza a busca no domínio do sistema (i.e. a *Web*) atrás de novas páginas a serem indexadas;
- Indexer – cria um índice de busca para as páginas recuperadas pelo *crawler* por este nó;
- Search and Administration interface – uma interface *Web* para administração do sistema e realizar buscas; todo nó é capaz de disparar buscas;
- Data Storage – persiste as estruturas de dados do sistema, geralmente na forma de tabelas *hash* distribuídas.

Do ponto de vista da criação de um sistema de busca universal e aberto (um dos sonhos do seu criador), o YaCy apresenta algumas vantagens e desvantagens:

1. Não há um servidor central, de tal forma que os resultados não podem ser censurados e a confiança do sistema é (ao menos teoricamente) maior;
2. Como é uma máquina de busca aberta e sem uma compania que seja sua dona, não há um mecanismo de propaganda centralizado;
3. Usuários mal intencionados podem criar bases fictícias, de forma que alguns resultados de busca sejam incorretos e/ou propaganda disfarçada.

A ScienceNet [5] é um projeto do Karlsruhe Institute of Technology que utiliza o YaCy como máquina de busca orientada a coleções de documentos científicos. Além de indexar domínios importantes da *Web*, como a Wikipedia, idéia do projeto é que cada universidade ou instituto de pesquisa ao redor do planeta contribua com sua produção científica na ScienceNet, reservando um ou mais computadores da instituição à rede contendo estes artigos e trabalhos.

Hoje, a ScienceNet é um sistema de busca com mais de 30 nós e 120 milhões de documentos entre páginas *Web* e trabalhos científicos. A ScienceNet vem crescendo a uma

velocidade de mais de 100 páginas por minuto. Pouco conhecido, o sistema ainda recebe pouco menos de 6 pedidos de busca por hora. E mesmo com uma carga de trabalho reduzida e com a maioria dos computadores envolvidos pertencendo a mesma universidade (ou seja, um sistema não tão geodistribuído e teoricamente menos suscetível a problemas de rede), nota-se as limitações do sistema: uma busca por 'search engine' levou quase 10 segundos[5] para retornar os 1,3 milhões de resultados.

2.5 Grid *Lucene*

Guardando muitas semelhanças com os sistemas P2P – como escala, geodistribuição e comunicação entre nós –, as grades computacionais são também um grande alvo de aplicações distribuídos. Elas são capazes de integrar grandes quantidades de sistemas heterogêneos por meio da Internet ou de redes corporativas de forma transparente, fornecendo grande poder computacional, geralmente se aproveitando dos recursos ociosos de seus integrantes [18] [22]. Dentre os problemas destes sistemas, está a dificuldade em saber *a priori* a disponibilidade dos nós, o que dificulta questões como o tratamento de confiança e *load balance*.

Bastante complexas e muitas vezes construídas inclusive sobre sistemas P2P, as grades vêem amadurecendo nos últimos anos e muitas aplicações vão sendo portadas para esses sistemas. Dentre as várias idéias de sistemas de indexação e busca para grades, destaca-se o GridLucene [28] por apresentar um protótipo funcional e por ser *open source*.

O desenvolvimento do GridLucene se deu em cima do Globus Toolkit 4 (GT4) [18], o *middleware* de grade mais conhecido e utilizado na atualidade e que está de acordo com a The Open Grid Services Architecture (OGSA). Dessa forma, o GridLucene pode se valer dos serviços já disponíveis dentro do *middleware* – como o GridFTP para troca de dados entre os nós. Mas mesmo com diversos serviços já disponíveis, o GridLucene ainda utiliza o Storage Resource Broker (SRB) para o armazenamento de dados dentro da grade. O SRB foi desenvolvido para funcionar sobre o Globus Toolkit e é capaz de armazenar petabytes de informação. O funcionamento do GridLucene é bastante simples:

- Coleção: a coleção de textos é armazenada no SRB na forma de blocos de documentos; estes blocos podem ser baseados em quantidade de arquivos, tamanho em bytes ou algum metadado dos documentos;

[5] consulta realizada em janeiro/2010

- Indexação: os nós pedem ao SRB por blocos de arquivos ainda não indexados, criam um índice para este bloco utilizando a biblioteca *Lucene* [2] e devolvem ao SRB o índice gerado;

- Busca: o nó que recebe os termos de busca pergunta ao SRB quais dos índices gerados são relevantes para esta busca e em que *hosts* estão localizados; a busca estão é feita a estes índices.

Os realizadores do trabalho executaram algumas baterias de testes de desempenho do GridLucene contra um índice *Lucene* local. Para a indexação, o GridLucene se mostrou entre 25 a 150% mais lento que um índice *Lucene* local, porém quanto maior a quantidade de documentos, menor é a diferença de tempo entre eles. Para a Busca, enquanto os tempos do *Lucene* local ficam na casa de 1 segundo independente da quantidade de documentos (do teste), no GridLucene o tempo chegou a até 40 segundos.

Várias ressalvas devem ser feitas sobre este trabalho:

- apesar de se tratar de um sistema (teoricamente) altamente distribuído, os testes parecem ser feitos com apenas 1 máquina remota;

- o *corpus*, apesar de ser uma base de testes conhecida e divulgada (INEX 2005[6]), é extremamente pequeno, contendo apenas 12.107 documentos; um sistema de busca real indexa coleções de milhões (e até bilhões) de documentos;

- a estratégia de utilização do índice é questionável: os índices indicados pelo SRB como relevantes não estão disponibilizados na forma de um *serviço*, mas sim acessados diretamente na forma de *stream* para ser tratado na máquina cliente. Dessa forma, cada consulta requer a transferência de partes do índice antes dele poder ser utilizado, o que acreditamos ser uma estratégia bastante ingênua e de baixíssima escalabilidade.

[6] http://inex.is.informatik.uni-duisburg.de/2005/

Capítulo 3

Índices Distribuídos

Apesar de serem duas as tarefas principais de um sistema de recuperação de informação a serem distribuídas (a manipulação dos índices e da coleção), são os índices a área mais crítica. Isso porque para o compartilhamento de arquivos existem diversas soluções distribuídas genéricas (como compartilhamento via NFS) e que resolvem – ao menos parcialmente, mas geralmente com baixa escalabilidade - o problema, enquanto que o tratamento de índices é muito mais complexo. Além disso, não existe dependência entre operações na coleção – os documento podem ser recuperados de diferentes máquinas -, enquanto que para os índices todos os termos buscados devem ser considerados e calculados. Assim, dependendo da organização dos índices distribuídos, as operações podem requerer que novos cálculos sejam realizados após obter respostas dos nós do *cluster*.

Neste capítulo, primeiramente serão discutidos aspectos básicos sobre um sistema de busca na seção 3.1 e então estratégias de índices distribuídos na seção 3.2.

3.1 Indexação

Antes de se discutir a distribuição dos índices, primeiro deve-se compreender o funcionamento da estrutura de dados não-distribuída mais utilizada, o arquivo invertido. Além disso, é necessário entender o processo de construção (a *indexação*) e utilização (a *busca*) dessa estrutura.

3.1.1 Arquivo Invertido

Diversas estratégias de indexação surgiram, como árvores de sufixo (e sua variação, vetores de sufixo) e arquivos de assinatura, mas foram os arquivos invertidos que se tornaram o padrão *de facto* para esta tarefa. Tais estruturas são constituídas por um vocabulário e um conjunto de listas invertidas. A figura 6.1 mostra uma representação visual de um arquivo

invertido simples.

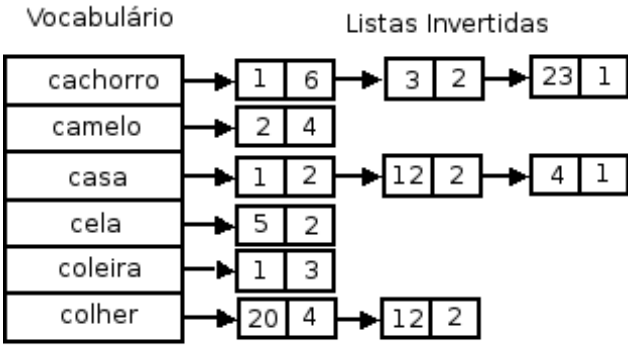

Figura 3.1: Representação de um arquivo invertido

O vocabulário é simplesmente uma listagem de todas as palavras únicas da coleção, em ordem lexicográfica, e cada palavra possui um ponteiro para a localização da sua lista invertida correspondente. Apesar do vocabulário parecer uma estrutura de dados muito grande, na verdade ele tem um tamanho bastante controlável: a tendência é que seu tamanho seja proporcional à raiz quadrada da quantidade de palavras da coleção [11].

A lista invertida de um termo registra informações sobre as ocorrências daquele termo do vocabulário na coleção. Em sua versão mais simples, as listas invertidas armazenam apenas listas do identificadores dos documentos que contém o termo, bem como as respectivas quantidade de aparições (frequência) nos documentos. Esta lista costuma ser ordenada pelo identificador do documento para facilitar operações de união e interseção entre duas listas. Muitas outras informações são comumente armazenadas, em especial a posição no documento de cada uma das ocorrências. Dessa forma, é possível localizar documentos onde duas palavras ocorrem exatamente uma depois da outra sem a necessidade de varrer todos os documentos onde ambas ocorrem.

O nome *listas invertidas* (e por consequência, *arquivo invertido*) se dá porque em um documento comum, vemos todos os termos que aparecem nele (documento ->termo), enquanto nas listas invertidas temos, para cada termo, a listagem dos documentos onde ele ocorre (termos ->documentos).

Muito do sucesso dessa estrutura de dados se dá devido a sua simplicidade e eficiência. A estrutura de dados é de tamanho inferior ao da coleção de texto (cerca de 30 a 60%),

tornando sua manipulação e carregamento viáveis. Já nas árvores e vetores de sufixos, os índices comumente possuem tamanho superior a várias vezes o tamanho da coleção.

3.1.2 Buscas

A busca em índices invertidos consiste em duas tarefas: (1) determinar quantos e quais documentos satisfazem uma dada consulta e (2) ordená-los segundo alguma lógica/algorítmo.

Para gerar a lista de documentos que satisfazem à consulta (ou *busca*), primeiramente o sistema varre o vocabulário atrás das listas invertidas de cada um dos termos da consulta. Se para uma busca com um único termo a lista invertida já responde quais documentos o contém, para buscas com múltiplos termos há de se levar em conta qual a operação booleana envolvida. Para uma operação *AND*, efetua-se a interseção das listas invertidas; para uma operação *OR*, efetua-se a união delas.

A seguir, o sistema deve ordenar essa lista de documentos, geralmente segundo algum critério que tenta estimar a *relevância* do documento à consulta. Outras ordenações podem ser realizadas por data do documento (ou outro metadado), pela importância do documento (como o PageRank usado no Google [13]) ou critérios híbridos. A mais tradicional forma de cálculo de relevância de um documento para uma dada consulta é através do *tf-idf* (*text frequency - inverse document frequency*) [12] [26]: o produto da freqüência do termo no documento pelo inverso da freqüência do termo na coleção.

Por exemplo, a palavra 'constituição' é uma palavra pouco frequente, então seu *idf* é bastante alto. Por outro lado, a preposição 'de' é uma palavra tão comum em qualquer coleção que independente de quantas ocorrências ela tenha num certo documento (*tf*), seu *idf* é tão baixo que essa palavra se torna pouco decisiva na ordenação dos documentos.

3.2 Índices Distribuídos

De maneira geral, todas as soluções de índices distribuídos compartilham semelhanças. Em especial, na separação entre máquinas que recebem solicitações e máquinas que efetivamente as processam.

As máquinas que recebem requisições são chamadas de *broker*; por conhecerem a organização do sistema distribuído, servem de *front-end* e redirecionadoras de requisições. O *broker* não possui nenhum pedaço do índice, apesar de muitas vezes armazenar o vocabulário

da coleção. Atrás do *broker*, no *back-end*, estão os *index servers*, as máquinas que realmente processam as consultas realizadas.

A seguir, os casos mais comuns de índices distribuídos serão apresentados, sempre considerando que o índice gerado para a coleção é grande demais para caber na memória principal de uma única máquina.

3.2.1 Índice Compartilhado

A idéia central por trás dos índices compartilhados, como pode ser visto na figura 3.2, é claramente inspirada nos sistemas de bancos de dados, onde diversas máquinas clientes acessam uma determinada base remota. Entretanto, os problemas de escalabilidade para soluções *single-node* continuam presentes, uma vez que apenas uma máquina continua sendo responsável por todo o índice da coleção.

As diversas requisições são reencaminhadas por um ou mais *brokers*, geralmente tendo origens diferentes. Por exemplo, um *broker* pode residir em um servidor *Web* que serve uma página de busca aos seus usuários, enquanto que outro *broker* pode disponibilizar buscas por meio de um serviço CORBA para a rede local. Assim, todas as requisições são afuniladas, muito possivelmente sobrecarregando o índice.

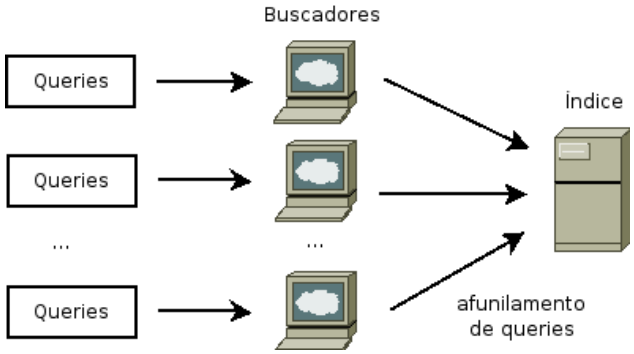

Figura 3.2: Representação de índices compartilhados

Ainda assim, essa arquitetura pode apresentar vantagens: a centralização das buscas como serviço numa só máquina tira a responsabilidade de outras; mas isso só faz sentido em situações onde o desempenho não seja uma questão crítica ou quando tem-se um baixo número de consultas ao longo do tempo. Sob o ponto de vista de sistemas de busca com alto desempenho, essa solução se mostra extremamente deficiente e ingênua.

3.2.2 Índice Replicado

Assim como nos bancos de dados, um primeiro sinal de evolução se deu com a utilização de múltiplas cópias da mesma base, aqui conhecidos como índices replicados. Apesar de ser uma solução extremamente simples, existe um bom ganho de desempenho, já que as buscas podem ser divididas entre todas as estações com réplicas. Dessa forma, espera-se uma carga de *1/N* para cada uma das *N* máquinas presentes. Cabe ao *broker* dividir as consultas, seja por um simples algoritmo de *round-robin* ou considerando pesos (por exemplo, uma busca com *N* termos é contabilizada como se a máquina tivesse recebido *N* buscas, para equilibrar as estações).

A figura 3.3 mostra a simplicidade da replicação de índice: uma das máquinas é considerada o indexador 'principal' e, ao final da indexação de novos documentos, seu índice é simplesmente copiado para todas as outras. Uma alternativa de melhor desempenho é a utilização do `rsync`[1], que repassaria apenas a diferença entre a estrutura de dados anterior do índice e a nova.

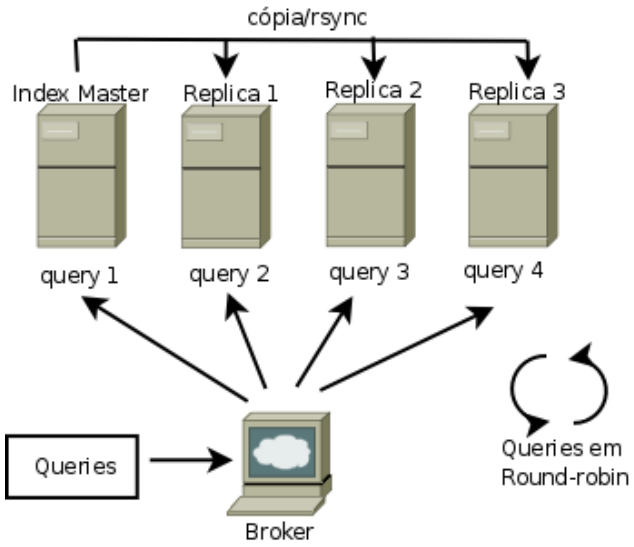

Figura 3.3: Representação de índices replicados

O sistema também tem uma boa escalabilidade, porque a carga pode ser dividida em um número arbitrário de estações, sempre com um *speedup* esperado próximo do linear (os custos de comunicação impedem uma aceleração verdadeiramente linear). Entretanto, o tratamento do índice em si continua sendo o mesmo de um sistema com um único nó, de tal forma que

[1] http://samba.anu.edu.au/rsync/

todos os problemas continuam e também são replicados: quando a coleção de textos cresce, o desempenho do sistema inteiro cai muito, uma vez que todas as máquinas estarão fazendo muitas operações de *swap*.

Analisando sob o ponto de vista das métricas de sistemas, nesta arquitetura o tempo de resposta de cada busca continua o mesmo de um sistema *single-node*, mas a vazão de buscas aumenta de forma próxima à linear. Algumas vezes o tempo de resposta desses sistemas é inferior ao de sistemas *single-node* e a vazão superior a linear (por exemplo, com 4 estações sendo 5 vezes mais rápidas do que uma só), mas geralmente isso acontece em função das máquinas do *cluster* estarem realizando paralelamente as operações de *swap* que o sistema original faria sozinho. Dos sistemas atuais, podemos destacar o Lucene/Solr [4] como utilizador essa estratégia.

3.2.3 Índices Locais

Os índices locais, também conhecidos por *divisão vertical* [9], possuem uma organização distribuída de complexidade média, sendo cada estação responsável por indexar apenas parte da coleção de textos, acarretando na criação de N índices pequenos com menos termos e (mais importante) listas invertidas menores. Isso torna o índice mais facilmente tratável e possível de ser armazenado inteiramente em memória principal. Como cada estação (e, conseqüentemente, o seu segmento de índice) conhece somente a sua parte da coleção de documentos, cada busca deverá ser retransmitida em *broadcast* para todos os nós do *cluster*, conforme a figura 3.4. Os resultados parciais de cada subcoleção de documentos serão reunidos e reclassificados (*merge*). Com vários índices pequenos, a escalabilidade do sistema aumenta bastante e o tempo de resposta por busca diminui (graças a redução de operações de *swap*), bem como aumenta a vazão do sistema.

Entretanto, essa escalabilidade está longe de ser linear para a adição de novas máquinas (como no caso de índices replicados), já que o sistema inteiro deve processar uma busca por vez (na verdade como cada nó pode ser *multi-threaded*, isso não é necessariamente verdade, mas como ainda assim todos os nós estão processando o mesmo conjunto de buscas, é uma simplificação válida).

Por outro lado, essa estrutura suporta de forma muito mais consistente o aumento da coleção de textos, já que esse fator de crescimento é dividido por entre as máquinas do *cluster* (quando M novos documentos são adicionados, cada uma das N estações adiciona

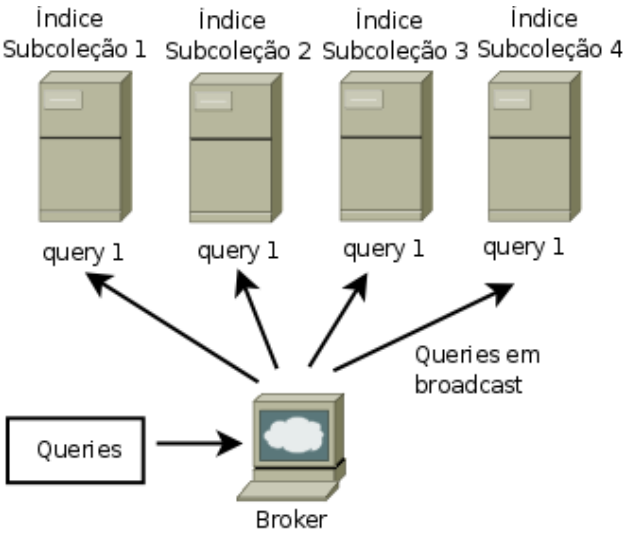

Figura 3.4: Representação de índices locais

M/N documentos a sua coleção), mantendo os segmentos de índices com tamanho reduzido. Cada índice local gerado possui um vocabulário de tamanho relativamente próximo ao da coleção completa (já que o crescimento do vocabulário segue uma função de raiz quadrada), entretanto são as listas invertidas que diminuem consideravelmente devido a menor gama de documentos. Considerando que todos os termos estão no eixo y e todos os documentos no eixo x, cada índice local seria a área representada por um intervalo no eixo x, daí o nome de divisão vertical.

Apesar de lidar com índices pequenos, as operações de balanceamento e de adição de novos nós para índices locais são extremamente custosas, precisando de uma reconstrução completa no índice: primeiramente a coleção de textos deve ser redividida e então o índice é recalculado, o que é um grande problema para sistemas em produção e que não podem ter o serviço interrompido. Os índices locais são a solução *de facto* para sistemas de busca [10], dentre eles o Lucene/Nutch [3], o Lucene/Solr [4] com sua *Federated Search* e o YaCy [6].

3.2.4 Índice Global

Os índices globais, ou *divisão horizontal*, são tidos como a estrutura distribuída mais complexa e paralelizável para índices, sendo desenvolvida a partir de análises sobre a estrutura dos índices locais. A principal diferença é que cada estação agora é responsável por um certo intervalo de termos do *vocabulário*. Analisando da mesma forma que nos índices locais, se todos os termos estão no eixo y e todos os documentos no eixo x, cada segmento do índice global seria a área representada por um intervalo no eixo y, daí o nome de divisão

horizontal.

Com essa nova divisão, os índices permanecem pequenos, mas as listas invertidas são do mesmo tamanho que no índice completo, o que pode dificultar um pouco a manipulação comparado com os índices locais. Entretanto, as vantagens são significativas para as buscas, já que as requisições não precisam mais ser repassadas em *broadcast* a todos os nós: somente aqueles nós que contém termos presentes na requisição serão envolvidos. Cada um deles recebe uma subconsulta diferente do *broker* (contendo somente os termos da consulta que o nó é responsável), que ao final deverá combinar os resultados de cada sub-consulta e ordená-los.

Como nem todos os nós estão envolvidos em cada consulta (como o exemplo da figura 3.5), eles estão livres para processar outras consultas em paralelo, possibilitando um aumento de vazão do sistema. Quanto maior for a relação entre a quantidade de estações no *cluster* e o número médio de termos em uma busca, maior será a quantidade de estações livres para outras consultas, o que mostra que o sistema possui grande escalabilidade. O crescimento da coleção de textos também tem impacto reduzido nesta organização, já que todos os termos indexados dos novos documentos estarão espalhados pelos nós do sistema.

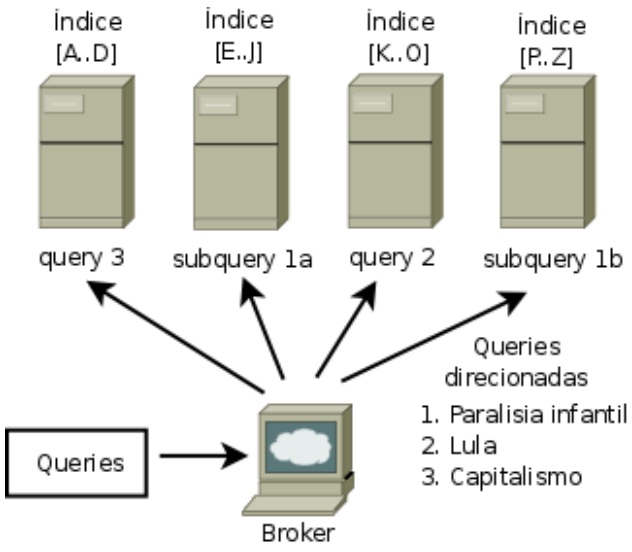

Figura 3.5: Representação de índices locais

A criação dessa estrutura é feita em 3 etapas [31]:

- Etapa 1 – Índices Locais: cada estação participante deve criar o índice local da coleção pela qual é responsável;

- Etapa 2 – Vocabulário Global: o *broker* recolhe de cada estação uma listagem de todos os termos indexados e o tamanho de sua lista invertida; essas listagens são unidas de forma a criar um vocabulário global e a quantidade de ocorrências de cada termo; o *broker* então faz uma partição do vocabulário pela frequência de cada termo e a quantidade de máquinas no sistema;

- Etapa 3 – Índices Globais: cada estação recebe do *broker* um intervalo de termos pelo qual será responsável e pede das outras estações as listas invertidas destes termos; cada estação cria um novo índice unindo as listas invertidas dos outros nós.

Nesse modelo, o papel do *broker* se torna muito maior do que nos modelos anteriores. Nos índices locais, o retorno de cada nó é uma resposta completa e pronta para ser utilizada, mas antes disso elas são unidas e reordenadas. Já nos índices globais, os resultados dos nós ainda sofrerão as operações booleanas (ou outras mais complexas) ao chegarem no *broker*. Por exemplo, uma busca por "paralisia AND infantil" pode ser divida em duas sub-consultas: "paralisia" indo para uma máquina e "infantil" indo para outra. Quando as listas invertidas dos dois nós chegam ao *broker*, ele ainda precisa realizar a interseção (AND) nas listas e então ordenar esta listagem para se considerar a resposta final.

A manutenção do índice global é outro fator a ser considerado. Para o índice global estar sempre equilibrado entre os nós, deve ser realizada uma reindexação completa da coleção de documentos. Indexações incrementais vão progressivamente desbalanceando o índice, requerendo reindexações periódicas completas. Os vários trabalhos a respeito de índices globais [13] [9] [31] [30] não discutem a manutenção do índice, como se o desbalanceamento fosse pouco problemático ou a coleção fosse sempre estática.

Poucos sistemas parecem utilizar essa organização de índices, como o Google e o buscador TodoBR [29] (adquirido pelo Google), mas sistemas como o Lucene/Solr planejam (eventualmente) sua implementação.

3.2.5 Breve Discussão entre Índices Locais e Globais

Por serem as duas estratégias mais desenvolvidas, é valida uma discussão sobre as vantagens de cada uma das abordagens.

- *Broker*: para os índices locais, o *broker* não é um ponto de gargalo [10], uma vez que ele trabalha somente em memória RAM e com tarefas de baixo consumo de CPU. Já nos índices globais, a necessidade de realizar operações mais pesadas no momento

de se compor os resultados parciais e em grandes quantidades de dados, se torna um potencial ponto de gargalo do sistema;

- Tamanho do *cluster*: *clusters* maiores beneficiam mais a arquitetura de índices globais, uma vez que as consultas tenderão a se espalhar pelo *cluster*, com cada máquina processando poucas consultas, enquanto todas as máquinas processam todas as consultas na arquitetura de índices locais;

- Divisão de Carga: tende a ser superior nos índices locais, onde estatisticamente todas as máquinas tendem a ter índices e listas de mesmo tamanho; os índices globais são muito suscetíveis a máquinas muito mais ocupadas que outras porque seu intervalo de termos é mais requisitado que o das outras (como termos iniciados com a letra 'a');

- *Disk Seeks*: como apenas um número limitado de máquinas recebem requisições para uma dada consulta, a arquitetura de índices globais realiza menos operações de *seek* de discos (uma operação 'lenta' e que reduz a longevidade dos discos);

- Retornos ao *broker*: nos índices locais, cada nó retorna seus melhores r resultados, onde r é a quantidade de resultados que o *broker* selecionará; nos índices globais, cada nó deve retornar muito mais resultados do que r: como as operações booleanas só serão realizadas no *broker*, os nós devem enviar mais resultados para se precaver daqueles que não satisfarão as operações;

- Desempenho: os experimentos em [27] mostram que índices locais são eficientes para resolver consultas *AND*, enquanto que os índices globais são mais eficientes para consultas *OR*.

- Escalabilidade: os experimentos em [9] nos parecem pouco extensivos e conclusivos – foi utilizada uma base de textos um tanto reduzida (de 2GB de tamanho) e distribuída em 4 máquinas; mais máquinas devem ser utilizadas para se avaliar limites da distribuição e impacto dos custos de comunicação;

Capítulo 4

Análise de Sistemas Atuais

4.1 O Google

Poucos ousariam negar que o Google é o maior sistema de busca da atualidade. Seu buscador indexa e armazena alguns bilhões de páginas, praticamente toda a superfície da *Web* (mas apenas uma pequena parte da *Web* profunda, mais difícil de se obter por depender de acesso privado ou por não ser referenciada por outras páginas). Em menos de 10 anos, o Google deixou de ser um projeto universitário para se tornar em um negócio de mais de US$150 bilhões de dólares e um dos alicerces da *Internet*.

Dessa forma, por mais que a equipe do Google esporadicamente publique artigos descrevendo o funcionamento do seu buscador, sempre devemos considerar que não estamos lidando com versões completas dessas soluções, afinal esse é o grande 'mapa da mina'. Se na primeira versão em 1998 as buscas demoravam entre 1 e 10 segundos [13], hoje em dia é difícil elas ultrapassarem alguns décimos de segundo.

4.1.1 Estrutura do Índice

O primeiro e último artigo do Google [13] que trata sobre o seu indexador remonta a 1998, quando ainda se tratava de uma pequena empresa tentando buscar espaço no mercado, portanto, é improvável que sua estrutura seja a mesma até hoje.

Como esperado, o índice é baseado em arquivos invertidos, cujos componentes são chamados de Lexicon (o vocabulário, que possuía 14 milhões de termos em 1998) e Hit Lits (as unidades das listas invertidas). O lexicon é constituído por uma estrutura que concatena todas os termos do vocabulário (separados por *null*) e por uma tabela hash contendo ponteiros para as posições dos termos.

Cada *hit list* é uma estrutura comprimida (por manipulação de bits – ver o apêndice B) em 2 bytes que armazenam as características de cada ocorrência de termo, como:

- Tipo: termos do corpo do texto (*plain hits*) são diferenciados dos termos especiais, como aqueles de títulos, *anchors* e URLs (*fancy hits*) – que possuem peso significativamente maior;
- Estado: se a fonte tem tamanho alterado, negrito, itálico ou alguma outra modificação. Palavras destacadas no texto são bastante representativas e por isso têm maior peso para buscas;
- Posição: a posição da ocorrência dentro do texto – até a posição máxima de 4096 (rótulo usado para ocorrências em posições superiores).

Os índices são armazenados em 64 segmentos de índices chamados de *Barrels*, com cada um armazenando um certo intervalo de termos, o que a nosso ver caracteriza a utilização de *global indexes*. A indexação em *barrels* é realizada em duas etapas, primeiro criando os *forward indexes barrels* que posteriormente se transformarão em *inverted index barrels*.

Os *forward index* são arquivos invertidos ainda não finalizados. Construídos documento por documento, são estruturas do tipo documento ->termo. Durante a construção dos *forward index barrels* também é realizada a construção do *Lexicon*, uma vez que todos os termos se tornam conhecidos.

O *Lexicon* então é partido pelos *barrels* que iniciam a ordenação dos seus *forward indexes* e as trocas com os outros, transformando-se em *inverted index barrels*. As listas invertidas são ordenadas pelo identificador do termo, com a ordenação secundária sendo feita por identificador do documento, assim o cálculo da interseção entre listas invertidas de diferentes termos é feita mais rapidamente.

Embora o artigo não cite explicitamente, não é difícil imaginar que a seguir esses *barrels* sejam distribuídos em diferentes estações (e certamente com múltiplos níveis de redundância), obtendo maior desempenho e escalabilidade no sistema. Com diferentes estações sendo responsáveis por diferentes segmentos do índice e cada segmento possuindo apenas um intervalo de termos do índice completo, podemos dizer que essa estrutura do Google era baseada na utilização de índice global (como detalhado na seção 3.2.4).

4.1.2 Indexação: Map/Reduce

Mantenha ainda ou não essa estrutura interna, em um artigo mais recente (2004) [17], o Google revelou (de forma um tanto velada) que a construção dos índices invertidos atualmente é realizada do Map/Reduce. O Map/Reduce, desenvolvido dentro do próprio Google, é tanto um modelo de programação distribuída facilitada quanto infraestrutura para execução paralela e distribuída. A idéia central do modelo é dar ao programador uma camada de abstração para a implementação paralela e de alto desempenho de uma série de tarefas com perfis semelhantes em diversos sistemas do Google. Todas essas tarefas possuíam uma quantidade enorme de dados (geralmente em forma de grandes arquivos) de entrada, de onde os dados relevantes eram selecionados, retirados e a seguir processados, de forma que o resultado final fosse uma série de relatórios ou dados organizados. Uma vez generalizado, esse modelo retiraria do programador as preocupações de comunicação, trocas de dados, paralelismo, escalabilidade e de como e onde o código será executado. Para os programadores utilizando esse modelo, o código parece completamente comum, exceto pela necessidade de definição de duas funções: *map* e *reduce*.

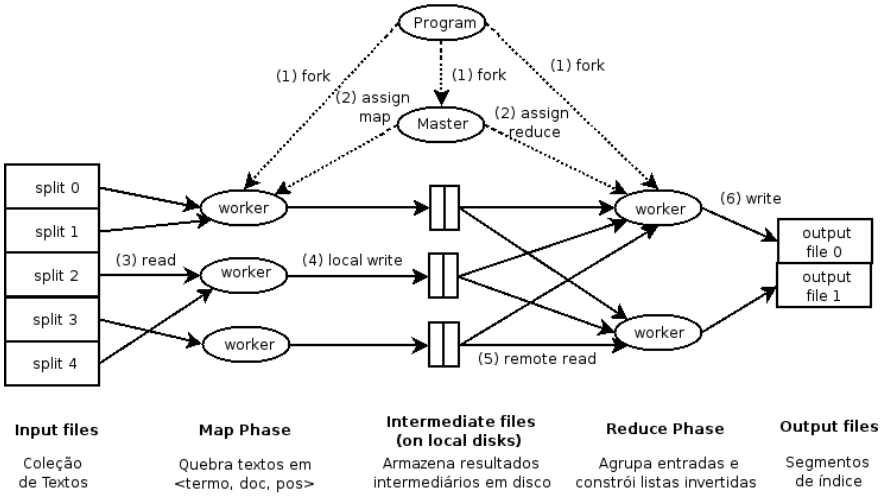

Figura 4.1: O processo Map/Reduce

Baseadas nas funções de mesmo nome das linguagens funcionais, a função definida como *map* deve ser capaz de varrer os dados de entrada e selecionar aquelas informações relevantes ao sistema. A saída da função *map* serão os dados de entrada para função definida como *reduce*, que tem o papel de sintetizar (geralmente, agrupar, filtrar ou somar) esses dados pré-processados e devolver ao sistema. As funções *map* e *reduce* são justamente os pontos onde o ambiente fará a distribuição e paralelização automática do código, totalmente transparente ao programador. Assim, os dados de entrada são divididos em N estações, cada uma realizando *map* sobre esses dados que a seguir serão reutilizados em M estações (muito

provavelmente diferentes daquelas que fizeram *map*) na função *reduce*.

Embora esse modelo não seja de ajuda nas buscas distribuídas, ele é utilizado na fase de indexação dos documentos, um problema que se encaixa justamente na categoria do modelo: lêem uma enorme quantidade de arquivos (os documentos) onde dados serão selecionados (fase *map*, a identificação de termos indexáveis e a construção das *hit lists*) e depois devem ser reunidos (fase *reduce*, onde esses termos e *hit lists* seriam divididos em *barrels*). A figura 4.1 demonstra a utilização do Map/Reduce para a indexação.

4.1.3 Armazenando a coleção: BigTable

Apesar de crítico, o desempenho de sistemas de busca de larga-escala, como o Google, não é determinado apenas pelo desempenho do índice de termos. Outros subsistemas são também igualmente críticos. Ao final da busca é gerada uma lista de identificadores de documentos (ordenados pela sua relevância para a busca), e ao usuário é exibida uma página contendo informações (*snippets*) uma sub-seqüência desses documentos. Essa página de exibição deve ser gerada após o término da busca, mas como qualquer termo do documento pode ser buscado, encontrar o trecho relevante de cada um dos documentos é uma tarefa que deve ser realizada *on-the-fly*. Dessa forma, a recuperação em alta velocidade dos documentos indexados também se torna uma tarefa crítica para o sistema, especialmente quando a coleção atinge a marca de vários milhões (ou bilhões) de documentos.

Por um lado, o simples armazenamento de milhões de documentos em um único diretório do sistema de arquivos supera em muito os padrões de uso e as otimizações do sistema de arquivos, tornando operações simples neste diretório extremamente lentas. Outra solução simples, o armazenamento dos arquivos em uma árvore de diretórios, possui maior velocidade de acesso. Entretanto ainda está longe das expectativas de sistemas de alto desempenho. Por outro lado, a utilização de sistemas de bancos de dados pode permitir algumas facilidades de distribuição e integridade dos dados, mas a um custo de desempenho e um maior nível de processamento. Assim, os sistemas de indexação e busca acabam por implementar bancos de dados de baixo nível e alto desempenho, específicos para o seu problema.

Na primeira versão do Google, em 1998, era utilizado um sistema chamado de BigFile [13], desenvolvido pela própria dupla de criadores da empresa. Esse sistema utilizava arquivos virtuais (alocados automaticamente pelo sistema) por meio de vários sistemas de arquivos e distribuídos via e referenciáveis por inteiros de 64 bits. A primeira parte desse sistema é

um repositório (*repository*), onde cada entrada é um documento diferente armazenado sob a forma de seqüências compactadas utilizando zlib chamadas de *packets*. Os *packets* contém o identificador do documento, os tamanhos da *string* de URL e do texto, além própria URL e do texto. A segunda parte desse sistema é um índice de documentos (*document index*) ordenados pelo identificador do documento que armazena as informações sobre o documento (estado, *checksum* e algumas estatísticas), além de um ponteiro para sua posição dentro do repositório. O índice de documentos também é responsável por realizar conversões do tipo URL ->identificador do documento. Por fim, a distribuição era garantida por meio de um simples esquema de NFS entre as estações que contém o repositório (ou partes dele).

Mais recentemente (2006), o BigFile evoluiu para uma complexa estrutura distribuída chamada BigTable [15]. Ao invés de simples referências para documentos de texto, a BigTable é uma matriz esparsa, distribuída, persistente, ordenada e multi-dimensional que armazena tipos arbitrários de dados (textos, imagens e outros) na forma de cadeias de bytes. A BigTable é capaz de armazenar diferentes versões dos dados, então caso a hora da versão desejada não seja passado, a versão mais recente é a recuperada. Assim, a localização de dados na BigTable acontece de duas formas:

- 1) (linha: string, coluna: string, hora:int64) ->byte[]
- 2) (linha: string, coluna: string) ->byte[]

A BigTable é ordenada lexicograficamente pela chave da linha e dividida em intervalos dinâmicos chamados de *tablets*, que se tornam a unidade básica de distribuição e balanceamento. O acesso à BigTable se dá por meio de uma API desenvolvida, com várias facilidades de uso: uma BigTable pode ser tanto utilizada como formato de entrada quanto de saída para computações utilizando o Map/Reduce.

Cada estação que possua *tablets* possui um TabletServer em atividade, capaz de receber consultas, adicionar e remover dados das suas *tablets* e, caso alguma tablet cresça demais, partí-la em duas *tablets* menores, possivelmente delegando uma das metades para outro TabletServer. O controle de acessos e escritas nas *tablets* é feito pelo Chubby [14], um sistema de *lock* distribuído leve e de baixo *overhead*. Para acessar uma tablet remota, primeiramente o cliente deve contatar a *root tablet* que armazena as *metadata tablets* que conhecem as informações (como localização, intervalo de linhas e tamanho) das *tablets* que referenciam.

Para um melhor funcionamento, o sistema da BigTable possui ainda uma série de otimizações, como *caching*, compressão, estruturas *in-memory* e leituras *read-ahead* (repasse de informações adicionais àquelas desejadas e que possam ser utilizadas futuramente), mas nenhuma delas tem seu funcionamento satisfatoriamente bem explicado.

4.1.4 Sistema de arquivo distribuído: Google File System

A utilização de *commodity hardware* para um sistema de alto desempenho sempre implica na necessidade de se lidar com os diversos problemas que podem (e inevitavelmente vão) acontecer neste hardware comum, em especial, as falhas nos componentes: discos, memórias, rede e até mesmo o suprimento de energia. Além disso, muitos problemas podem ser causados por falhas nas aplicações ou no sistema operacional, ou mesmo por falha humana. Assim, o monitoramento, a detecção de erros, tolerância a falhas e recuperação automática devem estar integrados ao sistema.

Os sistemas de indexação e busca também possuem características distintas de muitos sistemas no que tange a utilização de discos. Uma vez escritos, os arquivos são normalmente modificados apenas com escritas em *append* (não se modifica o conteúdo de um documento já indexado) e remoções de arquivos são extremamente raras (se existentes). O tamanho dos arquivos costuma ser considerável, já que ou os dados são muito grandes (como imagens de satélite do Google Earth) ou estão agrupados em grandes blocos (como milhares de documentos de texto plano armazenados dentro de *tablets*).

Além das possíveis falhas e características dos dados armazenados e do hardware, os arquivos ainda precisam possuir características de alto-desempenho para sistemas distribuídos, incluindo acesso concorrente, alta vazão de dados e minimização de utilização da rede. Dessa forma, todas essas características foram consideradas pela equipe do Google ao planejar e desenvolver o GFS (Google File System) [20] em 2003, um sistema de arquivos distribuído voltado para os sistemas da empresa, funcionando de forma integrada com outras ferramentas (em especial a BigTable).

Os *clusters* GFS são formados por um único mestre e um número qualquer de *chunkservers*, sendo acessado simultaneamente por diversos clientes. O mestre, sendo único, certamente se mostra um ponto de falha do sistema, por isso seu estado é constantemente replicado para múltiplas máquinas e, caso fique desabilitado, o DNS passa a apontar o nome de *host* do mestre para uma das réplicas. Isso porque a centralização apresenta vantagens

de coordenação e decisão sobre replicar blocos. É ele que detém um espaço de nomes (*namespace*) com todos os arquivos armazenados no GFS em qual *chunk* ele está armazenado. Assim, quando um cliente quer acessar um certo arquivo, ele primeiro pede para o mestre a sua localização, e só então vai até o *chunkserver* correto para acessá-lo. Quando um certo arquivo está replicado em diversos *chunkservers* (no GFS, padronizadamente em 3), o mestre decide qual deles o cliente usará, balanceando a carga do sistema.

Os *chunks* são os blocos de armazenamento de arquivos do GFS, que é implementado sobre o sistema de arquivos (embora não especifique qual) do Linux. Cada *chunk* é endereçado por 64 bits e possuem o tamanho fixo de 64 MB (16 mil vezes maior que os 4KB do tamanho geralmente utilizado pelo sistema de arquivos do Linux), assim, quando acessa grandes arquivos, o cliente não precisará fazer consecutivos requests pelos próximos *chunks* do arquivo desejado ao mestre. Outra vantagem é que como provavelmente diversas operações serão realizadas no mesmo *chunkserver*, reduz-se a quantidade de conexões TCP e que podem se tornar persistentes entre o cliente e o *chunkserver*. Por fim, um *chunk* desse tamanho reduz o tamanho dos metadados do GFS.

4.2 Apache Lucene

O Lucene [2] é uma biblioteca (não um sistema completo) *open-source* para indexação e busca em coleções de textos, desenvolvida em Java por Doug Cutting (inicialmente) no ano 2000 e tempos depois absorvida como subprojeto pela Apache Foundation para o projeto Jakarta, tendo uma base sólida de desenvolvedores e aprimoramento contínuo. Como se trata apenas de uma biblioteca, ele pode ser integrado a qualquer aplicação que precise de um sistema ágil de buscas.

Algum tempo depois da entrada do Lucene na Apache Foundation, ele foi promovido a um projeto próprio, e então diversos subprojetos Lucene surgiram, como *ports* para outras linguagens de programação, mas o mais importante deles era o Nutch, um sistema de recuperação de informação pronto para uso, escrito em Java baseado na indexação e buscas do Lucene. O Nutch é constituído por 3 partes: um *crawler*, que obtém páginas da *Web* para serem indexadas; a WebDB, um sistema de armazenamento (compactado) das páginas obtidas pelo *crawler*; e o Indexador, baseado em Lucene. Além desses 3 sub-sistemas, o Nutch também possui um serviço *Web* completo, incluindo página de busca e uma implementação de *result cache* (vide apêndice C). Entretanto, como Nutch não é uma solução para sistema de busca distribuído, ele vem sendo adotado apenas por diversos sites de busca pequenos ou

como um serviço residente de um site maior, como os do Universo Online, Creative Commons e JBoss.

Assim, iniciaram-se os esforços para tornar o Nutch uma solução de buscador *Web* de larga-escala e distribuído, utilizando muitas técnicas apresentadas nos artigos técnicos da equipe do Google (e apresentadas na seção anterior). Esses esforços de desenvolvimento passaram a se tornar mais genéricos e viraram um subprojeto próprio, a plataforma para computação distribuída Hadoop. Hoje em dia, o Hadoop já possui implementações próprias do Map/Reduce, do Google File System (aqui chamado de HDFS – Hadoop Distributed File System) e ainda está no caminho de uma implementação da BigTable (aqui chamada de Hbase) e do Chubby (chamado de ZooKeeper). Além disso, o HadoopStreaming permite a utilização do Hadoop (que é desenvolvido em Java) como plataforma distribuída para programas em outras linguagens.

A boa funcionalidade do Lucene, Nutch e Hadoop já chamou a atenção de grandes empresas, em especial do Yahoo! que recrutou diversos desenvolvedores desses projetos. Dentro do Yahoo! existe um *cluster* Hadoop com 902 nós bi-processados e armazenando 150 TB de informação, mostrando o valor do Hadoop como plataforma escalável e de alto desempenho para construção de um grande centro de processamento.

4.2.1 Estrutura do Índice

Os índices invertidos do Lucene são constituídos de um ou mais segments, e baseados em 3 estruturas fundamentais: o `Document`, que são as estruturas a serem indexadas (baseadas em um arquivo texto, mas não o próprio), é uma seqüência de `Field`s ; o `Field`, que diferencia os dados de um `Document` e diz o que deve ser indexado ou não, uma seqüência nomeada de `Term`s; e o `Term`, uma `String`.

Os segmentos de índice contém as seguintes informações:

- *Field Names*: o conjunto de Fields utilizados no índice. Como 'título', 'url', etc;
- *Stored Field Values*: para cada documento, uma lista <`Field`, valor>;
- *Term Dictionary*: o vocabulário do segmento;
- *Term Frequency Data*: a frequência de cada termo em cada documento do segmento;

- *Term Proximity Data*: as posições de ocorrência de cada termo em cada documento do segmento;
- *Normalization Factors*: para cada `Field` dos documentos, o peso de buscas naquele campo;
- *Term Vectors*: para cada `Field` de cada documento, um vetor <texto, frequência >(opcional)
- *Deleted Documents*: um arquivo opcional apontando os documentos deletados.

Para agilizar a recuperação de *strings* do disco, o Lucene armazena todas elas como uma cadeia de bytes, na qual no começo se encontra um *VInt* (o inteiro compactado por bytewise-compression da seção B) que diz quantos bytes a seguir devem ser lidos para formar a *string*. Para evitar o acesso sequencial, para cada arquivos do índice Lucene que possuam entradas de tamanho variável (graças a *strings* e *VInt*s), existe também um arquivo que contém os índices para as posições de cada entrada no arquivo.

Em uma consulta Lucene, primeiramente o *Term Dictionary* é varrido em busca do termo procurado, recuperando assim o seu term-id. Com os identificadores de todos os termos da busca, a *Term Frequency Data* (listas invertidas simples) podem ser consultadas para classificar os documentos mediante frequência dos termos nos documentos. Caso a busca por proximidade de termos (documentos onde termos diferentes apareçam em posições próximas são mais relevantes do que os em que apareçam distantes) esteja habilitada, também será consultada a *Term Proximity Data* (extensão da lista invertida simples, registrando as posições). A seguir, os pesos são multiplicados pelos valores em *Normalization Factors*: assim, buscas que encontrem dados em campos específicos possuirão um peso maior – assim os termos do título ou da URL podem possuir um peso maior do que termos do corpo do texto. Dessa forma, o classificador determinará a lista de documentos mais relevantes para a busca e caso os *Term Vectors* estejam habilitados, o corpo do texto dos documentos poderá ser recuperado. Caso não estejam, o sistema deve obtê-los de outra forma.

O Nutch, quando configurado para buscas distribuídas utiliza uma forma um tanto rudimentar para a utilização de índices locais (descritos na seção 3.2.3). Cada nó indexa a sua própria coleção local (dividida manualmente pelo administrador do sistema, e não automaticamente e de forma balanceada) com o Lucene/Nutch, e algum nó assume o papel de *broker*, recebendo consultas e as repassando em *broadcast* para os outros nós Nutch, utilizando o `MultiSearcher` do Lucene. É ainda um pouco longe de ser uma implementação eficiente e

escalável, mas certamente mais eficiente que um buscador *single-node*.

4.2.2 Criação de Índice: Map/Reduce

Como o Hadoop já implementa o modelo Map/Reduce (descrito na seção 4.1.2) de forma muito eficiente (e utilizada por diversos outros sistemas não-indexadores para tarefas pesadas, como o *website* Last.fm), a combinação Nutch/Hadoop é capaz de redirecionar a saída do *crawler* como entrada para a criação de índices distribuídos utilizando o Map/Reduce. A saída do sistema é um conjunto de segmentos do Lucene e que podem ser integrados ao sistema de busca. Entretanto, a absorção desses segmentos não é automatizada, cabendo ao administrador do sistema ordenar as fusões de segmentos segundo seu próprio bom-senso (e não balanceada pelo sistema). O Nutch ainda tem muito a melhorar nessa automatização e balanceamento de tarefas para um buscador distribuído.

4.2.3 A coleção: Hbase

A Hbase, uma implementação da BigTable do Google para o Hadoop, pode ainda não estar pronta para produção em alto desempenho, mas já possui uma organização e versões beta bem definidas. As buscas não são baseadas em uma linguagem como SQL e sim por meio de interfaces semelhantes a iteradores para se navegar entre as linhas e então se obter os valores das colunas. A Hbase é dividida em:

- HClient API: fornece ao cliente os métodos de acesso simplificado à Hbase;
- HRegion: um determinado intervalo de linhas. Equivalente às *tablets*;
- HRegionServer: servidor de HRegions. Equivalente aos *tabletservers*;
- HBase Master Server: o coordenador do sistema. Equivalente à *root tablet*.

Uma importante diferença entre a BigTable e a Hbase, é a ausência de um serviço de *lock* distribuído para a Hbase. O DistributedLockServer para o Hadoop também ainda não está pronto, portanto a Hbase deve utilizar outra forma para fazer os por *locks* e mapeamentos. Assim, a Hbase tem a META Table, que mapeia as informações HRegion ->HRegionServer, de forma semelhante (mas mais limitada) que o Chubby faz com Tablet ->TabletServer.

4.2.4 Sistema de arquivo distribuído: HDFS

O HDFS, uma implementação nos moldes do Google File System, objetiva alcançar o mesmo resultado, dando garantias de segurança, disponibilidade e integridade dos dados

distribuídos, enquanto otimiza a leitura de grandes arquivos e os replica para obter maior desempenho e balanceamento, escalável para milhares de usuários e estações.

Para o HDFS, um *cluster* Hadoop possui um NameNode e diversos DataNodes, nos quais os arquivos são armazenados em *blocks* (os *chunks* do GFS, e com mesmo tamanho de 64 MB), e os *blocks* são armazenados em DataNodes (os *chunkservers* do GFS), que os disponibilizam de forma distribuída. O NameNode (equivalente ao master do GFS), possui um espaço de nomes (*namespace*) que é capaz de fazer associações nomes de arquivos ->*blocks*. O NameNode também é quem decide para qual das réplicas deste *block* ele irá direcionar as requisições. Essas decisões são feitas com mapeamentos entre *blocks*, DataNodes e INodes, e podem ser da seguinte forma:

- *block* ->DataNode map (via classe `FSNamesystem`.blocksMap)
- DataNode ->*block* map (via classe `FSNamesystem`.datanodeMap)
- INode ->*block* map (via classe `INode`.blocks)
- *block* ->INode map (via classe `FSDirectory`.activeBlocks)

Outras características do HDFS:

- Operações de arquivo: open(), create(), close(), read(), append(), truncate(), delete(), rename(), undelete() e seek(), além de obter os números dos blocos e determinaro nível de replicação;
- Operações de sistema de arquivo: readdir(), rename() e para estatísticas (nfiles, nbytes, nblocks);
- *Check-pointing* e *journaling*
- *Lock*: acesso exclusivo ou compartilhado para leituras sequenciais/aleatórias e appends;
- Detecção e correção de erros automáticas;
- Operação append() atomizada;
- Noção da topologia do *cluster* para alocação de blocos mais eficiente.

Apesar de já bem implementado e totalmente funcional, o HDFS ainda pode ser aperfeiçoado para lidar melhor com algumas situações. Por exemplo, caso o *cluster* Hadoop esteja com o HDFS perto da saturação, sem mais espaço para armazenar dados, e novas

estações forem adicionadas a ele com o serviço de DataNode em funcionamento, ele não fará a redistribuição automática (embora os novos *blocks* – e suas réplicas – tenham chance superior de irem para estas máquinas). O administrador então pode escolher alternativas (talvez até automatizadas, via scripts) para conseguir o mesmo efeito, como:

- Selecionar um subconjunto de arquivos que ocupam bastante espaço e copiá-los sob para diferentes áreas do HDFS (forçando replicação nas novas máquinas). Depois, deleta-se as velhas cópias e se renomeia as novas cópias para os nomes originais. Entretanto, deve-se paralisar o serviço até que a tarefa esteja completa;

- Uma forma mais simples, e sem interrupção do serviço, é aumentar o nível de replicação global do sistema e, uma vez que as transferências terminem, reduzir novamente o nível de replicação, de forma que as réplicas a mais nos DataNodes sejam aleatoriamente removidas. Entretanto, é lento e pode não haver espaço o suficiente para replicar todo o sistema.

- Outra forma de fazer o balanceamento é desconectar os DataNodes mais cheios, esperar o sistema replicar os blocos perdidos para satisfazer o número de réplicas correto, e então trazer o DataNode de volta. As réplicas a mais serão aleatoriamente removidas. Esse método costuma ser o mais rápido e sem derrubar o sistema, mas pode ser necessário repeti-lo diversas vezes.

4.2.5 Apache Solr

O Solr é uma solução *open source* de *enterprise search* fortemente baseada no Lucene. É um serviço pronto para uso, rodando sobre o Apache Tomcat[1] ou outro *servlet container*, e com diversas funcionalidades úteis a um sistema de recuperação de textos na *Web*, como APIs de consulta via SOAP e JSON, *result cache* (ver o apêndice C.1, um sistema de *plugins* e uma interface *Web* para administração do sistema.

O fato de ser um serviço pronto para uso e baseado no Lucene fez a popularidade, uso e comunidade do Solr crescerem rapidamente, sendo utilizado como solução de busca interna para dezenas de *websites*[2] de *e-commerce* e conteúdo, como o *website* da Casa Branca dos EUA, a videolocadora Netflix, vários *subsites* da AOL (America On-Line) e o digg[3], uma das redes sociais de maior tráfego da *Internet*.

[1] http://tomcat.apache.org
[2] http://wiki.apache.org/solr/PublicServers
[3] http://www.digg.com

Atualmente, o Apache Solr tem duas soluções de índices distribuídos já desenvolvidas, uma por replicação e outra por índices locais, aqui chamada de *Federated Search*. Por ser um sistema *open source* largamente utilizado no mercado e com uma solução de distribuição por índices locais, o Apache Solr será utilizado como comparativo ao sistema desenvolvido neste trabalho no experimento na seção 6.4.3.

Capítulo 5

Arquitetura

O desenvolvimento de um sistema de busca distribuído e escalável é um dos objetivos deste trabalho, de tal forma que a arquitetura e decisões de projeto deste sistema devem ser apresentadas e discutidas. *A priori*, duas decisões já estavam tomadas: o Lucene [2] seria utilizado como base para o serviço de índices, por ser *open source*, de fácil utilização e largamente utilizado e testado pela comunidade; a (principal) linguagem de programação do sistema seria Java, por ser uma linguagem compilada, de ótimo desempenho e com muitas ferramentas e bibliotecas disponíveis, além de ser a linguagem na qual o Lucene é desenvolvido.

Neste capítulo serão discutidas as formas de comunicação entre os serviços do sistema na seção 5.1; o serviço de armazenamento e recuperação da coleção original, as Tablets, na seção 5.2; e como os índices distribuídos são criados e consultados, na seção 5.3.

5.1 Comunicação

A forma de comunicação entre os nós dos serviços é uma importante decisão de projeto. As opções são muitas, sendo Java RMI[1](JRMI), CORBA[2] e Web Services (SOAP[3] e XML-RPC[4]) as mais comuns. Alguns pontos deveriam ser considerados:

- *Cluster* – como a infraestrutura a ser utilizada neste projeto é basicamente uma rede local de computadores, não existe a restrição pela utilização de Web Services para comunição remota sem o bloqueio de *firewalls*; pode-se assumir que todas as portas estão liberadas para uso e que se está em uma rede dedicada ao sistema;

[1]http://java.sun.com/javase/technologies/core/basic/rmi/
[2]http://www.corba.org/
[3]http://www.w3.org/TR/soap/
[4]http://www.xmlrpc.com/

- Desempenho – a tecnologia de comunicação utilizada deve ter alto desempenho e de pouco *overhead* de dados transmitidos – um aspecto onde os Web Services deixam a desejar; experimentos em [24] e [23] mostram que JRMI e CORBA possuem um *overhead* inferior aos dos Web Services, além de maior desempenho, com ligeira vantagem para o JRMI;

- Desenvolvimento – a quantidade de ferramentas e bibliotecas disponível que dêem suporte ao desenvolvimento com velocidade nesta tecnologia são um fator desejável;

- Plataformas e Linguagens – não há planos de alterar a linguagem da plataforma desenvolvida, o que seria um ponto de corte para JRMI; é esperado que as consultas ao *front end* do sistema sejam possíveis por meio de diversas linguagens e tecnologias, mas isso não afeta a comunicação interna do *cluster*.

Devido a estes fatores, a opção de utilização pelo JRMI acabou sendo uma escolha muito natural.

5.2 Tablets

Apesar da literatura em sistemas de indexação e busca ser bastante extensa, raras são as vezes onde se trata de um elemento básico desses sistemas: a armazenagem das versões originais dos documentos indexados. A rápida recuperação dos arquivos da coleção é indispensável para o tempo de resposta das buscas. Após o sistema determinar aqueles documentos mais relevantes para a requisição, uma página de resultados deve ser elaborada e exibida ao usuário, contendo os k melhores documentos para aquela requisição. Nesta página, para cada documento relevante, mostra-se também um pequeno trecho de cada um (os *snippets*), destacando a ocorrência dos termos nos mesmos. Dependendo da forma de armazenamento dos arquivos, isso pode implicar na utilização de diversas operações de entrada/saída para abrir e recuperar o conteúdo destes arquivos, o que é um potencial gargalo do sistema. Soluções simplistas, como deixar todos os arquivos em diretórios ou armazená-los em bancos de dados tradicionais, extrapolam a capacidade dos sistemas de arquivos e possuem eficiência limitada.

Neste sentido, destaca-se, mais uma vez, o Google, cujas abordagens original e atual são descritas em artigos publicados, entretanto são não de código aberto. Originalmente, o Google utilizava um sistema chamado BigFile [13], um sistema de armazenamento baseado em ISAM (*Indexed Sequential Access Method*). Atualmente, o armazenamento é feito por meio da combinação do Google File System [20] e da BigTable [15], sendo o primeiro um sistema de arquivos distribuído de alto desempenho. e o segundo um banco de dados distribuído e

orientado a colunas.

Apesar da BigTable possuir um sistema equivalente *open source* construído segundo as mesmas diretivas, a HBase da plataforma Hadoop [1], durante o andamento deste trabalho ela foi repetidamente tida por seus próprios desenvolvedores como ainda não pronta para sistemas em tempo real, sendo apropriada para trabalhos em lote. Somente a partir da recente versão 0.20, ela passou a ser considerada para este propósito.

Esta seção descreve a arquitetura, os dois serviços que a constituem e as decisões de projeto das *Tablets*, um sistema de armazenamento de arquivos distribuído e baseado em ISAM. Experimentos de desempenho das *tablets* serão abordados na seção 6.3.

5.2.1 Conceitos

A arquitetura das *tablets* é fortemente baseada no BigFiles [13], com serviços mais bem definidos e distribuídos, permitindo escalabilidade ao sistema. Os documentos não ficam armazenados em diretórios ou SGBDs, e sim em *tablets*, estruturas de dados persistentes do tipo ISAM desenvolvidas para este projeto e que permitem a recuperação de documentos do disco em grande velocidade. O ISAM é dividido em cabeçalho e corpo. Enquanto no corpo os arquivos são armazenados como uma única grande sequência de bytes, no cabeçalho existem ponteiros para as posições de memória onde cada arquivo começa – e o final do arquivo é a posição inicial do próximo arquivo.

Apesar da literatura apresentar críticas e mostrar que índices ISAM são estruturas bem simples e antigas (do tempo das fitas magnéticas), quando se trata de sistemas de recuperação de dados de altíssimo desempenho notamos a presença de tais índices como as tabelas MyISAM do SGBD MySQL e o BigFile. Índices ISAM são especialmente recomendandos para sistemas de alto desempenho onde as operações sob a tabela sejam basicamente *select* e *append*. Dessa forma, as críticas ao modelo geralmente se dão devido à dificuldade em se realizar operações de *delete* e *update* (especialmente) nesse modelo.

Se uma operação de *update* tiver uma nova versão do dado com tamanho maior que o da antiga, será necessário realizar um deslocamento de toda a sequência de bytes à frente para comportar a nova versão; outra opção seria marcar a posição antiga como apagada e adicionar o novo dado noutra posição, ocasionando fragmentação. Da mesma forma, se a nova versão for menor que a versão antiga, também haverá fragmentação.

Uma vez que os dados sejam armazenados em poucos blocos ISAM, de tal forma que todos estes arquivos estejam sempre abertos durante a execução do programa, e os ponteiros residentes em memória, a principal vantagem deste modelo se torna a possibilidade de se recuperar um dado com apenas *uma* operação de entrada/saída. Utilizando simplesmente sistemas de arquivos ou SGBDs são necessárias múltiplas operações, sendo as primeiras para encontrar a posição do dado (revirando *inodes* de diretórios, por exemplo), então uma nova operação para abrir o arquivo e outra para ler o dado.

5.2.2 Tablets

Cada *tablet* é um bloco de dados em disco, limitado a 64 MB, como no BigFiles, ou 32.768 arquivos internos, muito embora ambos limiares sejam customizáveis. Para evitar deslocamentos de bytes quando novos arquivos são adicionados, o cabeçalho deve ter tamanho fixo e quanto mais arquivos sejam possíveis em cada *tablet*, maior é o cabeçalho. Documentos adicionados em sistemas de busca, basicamente textos, raramente serão menores que 2KB, de forma que 32.768 arquivos são um limite bem pessimista.

O cabeçalho das *tablets* é formado pelas seguintes estruturas:

- *FirstId* – o identificador do primeiro dado armazenado na *tablet*;
- *NumDocs* – a quantidade de dados armazenados nesta *tablet*;
- *Timestamps* – o horário de criação e de última atualização desta *tablet*;
- *Checksum* – o MD5 do bloco de dados da *tablet*, para garantia de integridade dos dados;
- *Offsets* – um vetor de ponteiros para posições de inícios de dados dentro da *tablet*;
- *Status* – um vetor de bits que marca o estado (normal ou apagado) de cada dado;
- *Namespace* – um vetor com identificadores de 'nome' de cada arquivo (na verdade, o MD5 do nome original do arquivo);

Dessa forma, o tamanho do cabeçalho de cada *tablet* tem o tamanho de 659.496 bytes; pouco menos de 1% do tamanho total da *tablet* para os valores *default* de tamanho e quantidade máxima de arquivos. Ao contrário do BigFiles onde cada arquivo interno é identificado por um inteiro de 64 bits, em *tablets* eles são identificados por inteiros comuns (32 bits). Como o limite de um inteiro comum é de mais de 4 bilhões de possibilidades, não

foi visto como necessária a utilização de inteiros longos como identificadores. O identificador numérico de cada dado armazenado é chamado de *tabid*.

5.2.3 TabletZones e o Serviço de Controle

As TabletZone são grupos lógicos de *tablets* que contém um determinado intervalo de identificadores. Cada zona é responsável por 65.536 *tabids*, de tal forma que se a zona está completa, existe no mínimo duas *tablets* dentro dela. O limite de endereçamento do sistema é de 65.536 zonas, assim, os primeiros 16 bits de um *tabid* identificam a zona em que ele está armazenado.

A principal função das zonas é de *reserva de endereçamento*. O *tabid* é único no sistema, mesmo para *tablets* distribuídas, o que significa que deve haver um controle durante a criação das *tablets* – função do serviço de controle. As zonas então são utilizadas para se evitar que cada nova operação de inserção exija uma consulta ao serviço de controle para se descobrir qual o próximo *tabid* disponível – uma vez que o serviço de controle reserva uma zona, não são necessárias novas consultas até que aquela zona esteja completa. Essa economia de consultas pode parecer irrelevante para um serviço local, mas é fundamental para um sistema distribuído.

O serviço de controle é bastante simples, basicamente um *broker* que (1) conhece a localização dos dados armazenados no sistema, (2) é capaz de realizar mapeamentos de nomes ->*tabids* e (3) reserva *tablet zones* para o serviço de dados. Normalmente, cada nó participante do sistema terá uma zona ainda incompleta, na qual adiciona seus novos documentos. Quando esta zona está completa, o serviço de dados deve requisitar ao serviço de controle uma nova zona. Quando o serviço de controle recebe um pedido por um certo *tabid*, ele deve primeiro descobrir a qual zona aquele *tabid* pertence (isolando os primeiros 16 bits do *tabid*) e então redirecionar o pedido para a máquina que possui o serviço de dados que serve aquela zona.

Finalmente, outra utilidade das zonas é servir como unidade de distribuição. Quando novas máquinas são adicionadas ao sistema, o serviço de controle é capaz de dividir as zonas para a nova quantidade de máquinas. Apesar de ainda não implementado, a intenção é armazenar as zonas no HDFS (vide seção 4.2.4) do Hadoop, de forma que o sistema consiga também se reestabelecer e reequilibrar mesmo em caso de quedas de máquinas integrantes do sistema.

5.2.4 Serviço de Dados

O serviço de dados é o responsável pela manipulação e acesso do conteúdo das *tablets*. Cada *host* que tenha um serviço de dados no ar é responsável por servir as *tablets* de uma certa quantidade de zonas: seja para adicionar novo conteúdo, seja para recuperar conteúdo armazenado.

Quando recebe um novo arquivo para armazenar, o serviço deve primeiramente realizar duas verificações: se existe algum *tabid* disponível nas zonas que gerencia e se tem alguma *tablet* ainda não completada. Caso não haja *tabids* disponíveis nas zonas gerenciadas, o serviço deve requisitar uma nova zona ao serviço de controle e caso não haja nenhuma *tablet* incompleta, o serviço deve criar uma nova *tablet*. Quando essas verificações estão completas, o serviço realiza um *append* na área de dados da *tablet*, adiciona a posição inicial deste no vetor de *offsets* e o MD5 do nome do arquivo no vetor de *namespace*.

Uma variação desta situação é quando o serviço de controle percebe que o novo arquivo pedido para ser armazenado possui o mesmo nome que outro já existente no sistema. Neste caso, o serviço de controle descobre qual zona e máquina estão responsáveis por este conteúdo e pede a este serviço de dados que realize um *update*. Primeiramente, o serviço deve marcar o *tabid* como deletado. A seguir, procurar na listagem de *tabids* deletados pelo que possua tamanho mais próximo do novo conteúdo (política *best fit*, para minimizar a fragmentação) e então substituir o conteúdo antigo pelo novo (com o espaço sobressalente preenchido por caracteres de espaço em branco), além de atualizar o nome do arquivo. Caso não haja nenhum *tabid* com tamanho suficiente para ser reaproveitado pelo novo conteúdo, ele é adicionado como um novo arquivo.

O serviço de dados também suporta filtros para o conteúdo armazenado que podem ser configurados na sequência desejada, como o padrão de projeto *Chain of Responsability* [19]. Atualmente os filtros disponíveis são os mais comuns:

- Cache – um sistema de *cache* simples, com política LRU (menos recentemente usado) – caso o *tabid* requisitado tenha o conteúdo no *cache*, retorne-o; caso contrário, consulte o ponteiro, recupere do disco, armazene no *cache* e então retorne o conteúdo solicitado;
- Compressão – compacta o conteúdo antes de adicionar na *tablet*; quando solicitado, descompacta antes de retornar;

- Logging – registra os acessos ao conteúdo;

A ordem dos filtros faz toda a diferença: se o filtro de logging estiver na frente, todas as requisições são sempre registradas; se estiver por último, somente registra as requisições que não foram respondidas pelo serviço de *cache*.

5.3 Índices

5.3.1 Estrutura

Na seção 3.2.5 são feitos comparativos entre as estratégias de índices locais e índices globais. Para a seleção da estratégia, outros pontos devem ser avaliados:

- Manutenção do Índice – um problema pouco relevante para experimentos, mas muito significativo para um sistema em produção: os índices devem ser criados de forma incremental e bem equilibrada; reindexar uma base grande pode levar muitas horas, uma operação caríssima para um sistema em produção; este é um ponto negativo para os índices globais;

- Desempenho – tanto [27] quanto [9] não são definitivos quanto a superiodade de uma das duas estratégias na grande maioria das situações; aparentemente, os índices globais se tornam claramente superiores conforme o *cluster* aumenta de tamanho; entretanto, neste trabalho não contaremos com um *cluster* grande, conforme será discutido no capítulo 6;

- Balanceamento de Carga – apesar de não encontrarmos estudos específicos a respeito, podemos imaginar que o balanceamento de carga da utilização dos termos não é ideal: alguns intervalos de termos podem ser muito mais consultados do que outros; assim, na estratégia de índices globais, a carga de algumas máquinas é potencialmente superior a de outras (por exemplo, a máquina que contenha o intervalo de termos iniciados com a letra 'a';

- Padrão da Indústria – os índices locais são o padrão *de facto* da indústria [10] de máquinas de busca;

- Desenvolvimento – a estratégia de índices globais é contrária a organização interna do Lucene; deste modo, a implementação de tal estratégia implicaria em modificar grande parte do código, o que faria necessária novas reimplementações a cada nova versão do Lucene;

Devido a estes pontos, foi tomada a decisão de se desenvolver o sistema com índices distribuídos locais. Adicionalmente, neste trabalho, investigaremos uma variação dos índices locais: o *corpus* de cada estação será indexado em não apenas um índice, mas divido em múltiplos índices menores (que chamaremos de *segmentos*), os quais terão réplicas armazenadas em um sistema de arquivos distribuído, como o HDFS do Hadoop. As vantagens dessa alternativa estão relacionadas à possibilidade do sistema se reequilibrar rapidamente em caso de quedas (tolerância a falhas) e ou adição de novas de máquinas (sistema escalável) segundo orientação do *broker*. Caso novas máquinas sejam adicionadas ao sistema, o *broker* indicaria quais segmentos essas máquinas devem obter do HDFS para se tornarem responsáveis e comunica aos antigos responsáveis para não mais disponibilizarem esses segmentos. Caso o sistema note perda de máquinas, o *broker* deve distribuir a responsabilidade daqueles segmentos 'perdidos' para as máquinas ainda disponíveis (que obterão esses segmentos perdidos também do HDFS). Dessa forma, se cada máquina trabalhar com uma quantidade grande de segmentos de índice, isso garante um sistema com carga melhor distribuída após o reequilíbrio. Entretanto, o desempenho do sistema cai para maiores quantidades de segmentos servidos em cada estação. Isso porque, para a uma máquina, procurar em um índice grande é mais rápido que procurar em diversos índices pequenos e juntar os resultados.

5.3.2 Indexador

O indexador é um *pipeline* constituído por diversas *threads* pelas quais a coleção de textos deve passar para completar a indexação. Cada *thread* representa uma etapa diferente, com suas premissas e responsabilidades para com o sistema. As *threads* que compõe o indexador hoje são:

- Source: responsável por obter os novos arquivos a serem indexados da fonte e os carregar para a memória, normalmente um diretório local específico; centraliza as operações de leitura;
- Parsers: transforma um arquivo lido do disco em um `Document` do Lucene; preenche os campos de metadados, realiza limpeza de caracteres inválidos para indexação e *tags* HTML (ou outras programadas para serem retiradas);
- Tableter: verifica se o novo documento já está armazenado e marca para atualização ou inserção; atualiza/insere o conteúdo original nas *tablets*;
- Indexers: cada *thread* indexadora cuida de um índice diferente; para cada `Document` retirado da fila, são realizadas as operações de retirada de acentos e *stopwords*, tokenização do texto e então a criação do arquivo invertido;

- IndexMerger: unifica os índices gerados pelas *threads*, caso se queira otimizar para um único índice (fase opcional);
- TrieGenerator: realiza um *dump* do vocabulário dos índices, os unifica e gera a *trie* do vocabulário (vide Apêndice A).

Deve-se destacar que, graças a estratégia de índices locais, o indexador desconhece que faz parte de um sistema distribuído – as etapas são feitas sem necessidade de coordenação ou sincronização com outras máquinas. A única exceção se dá ao sistema de *tablets* que deve pedir reserva de novos *tablet zones* ao *broker*, uma operação de baixo custo que só acontece a cada 32.768 documentos a serem indexados.

As etapas do *pipeline* são inteligadas por filas de objetos de tamanho limitado, de tal forma que as *threads* assumem papéis de produtoras, consumidoras ou ambos. Quando uma *thread* no papel de produtora nota que a fila de saída está cheia, ela se bloqueia; quando uma *thread* no papel de consumidora nota que a fila de entrada está com poucos elementos (30% do tamanho máximo, valor customizável), ela envia um sinal para *thread* produtora retornar a atividade. Dessa forma, espera-se manter todos os processadores da máquina sempre ativos. Quando uma *thread* produtora nota que não tem mais itens a serem processados, ela 'envenena' sua fila de saída. Uma fila envenenada que não contém mais itens sempre retorna um objeto de 'veneno'. Esse padrão, no qual o aviso de encerramento de uma *thread* se dá por meio de um objeto especial vindo de filas, é conhecido como *Poison Pills* e descrito em [21]. Objetos de fim pré-estipulados são mais seguros e representativos do que a utilização de referências nulas.

De forma a simplificar a programação de *threads* para o indexador, todas devem ser extensões da classe abstrata `FlowThread`, que faz parte da API deste trabalho. Os 4 métodos da API que podem ser definidos, são:

`setUp()` – primeira tarefa realizada pela *thread* após ser criada;

`tearDown()` – última tarefa realizada pela *thread* antes de morrer;

`processItem()` – detalha como processar um item da fila;

`doOwnWork()` – método executado por *threads* que não usam filas;

Uma implementação de `FlowThread`, caso necessite, também deve apresentar uma nova implementação do `AbstractBuilder` – do padrão de projeto *Builder* apresentado em [19].

Por meio da construção da *thread* pelo *builder*, o sistema saberá localizá-la corretamente no *pipeline*. Os 5 métodos pré-definidos no `AbstractBuilder` são:

`inputFrom(Queue)` – informa que esta *thread* consome itens de uma dada fila;

`outputTo(Queue)` – informa que esta *thread* produz itens para uma dada fila;

`startsAfter(FlowThread)` – determina que esta *thread* só inicia após o fim *thread* informada;

`priority(int)` – muda a prioridade desta *thread*;

`build()` – constrói efetivamente a *thread*.

A listagem a seguir exemplifica a criação de um pequeno *pipeline* com 3 *threads*, dos tipos fictícios `Thread1` (produtor), `Thread2` (consumidor) e `Thread3` (que só pode iniciar após o fim da `Thread2`), todos implementações de `FlowThread`:

```
Queue queue = new Queue("source-parser", queue_size),

FlowThread t1 = new Thread1.T1Builder().
                    .outputTo(queue)
                    .priority(Thread.MAX_PRIORITY)
                    .build();
FlowThread t2 = new Thread2.T2Builder()
                    .inputFrom(queue)
                    .build();
FlowThread t3 = new Thread3.T3Builder()
                    .startsAfter(t2)
                    .build();
```

A definição de `FlowThread` retira do programador a necessidade de compreender e trabalhar em baixo nível com diretivas de sincronização, sendo necessária apenas a programação da função daquela etapa do *pipeline*. Além das *threads* definidas neste trabalho, outras que estão fora do nosso escopo já foram implementadas e estão em produção – a tarefa de seus programadores foi a implementação dos métodos da API. Além disso, a utilização de *Builders* para essas *threads* torna o código do *pipeline* bastante simples e legível.

5.3.3 Buscador

A utilização da estratégia de *índices locais* e funcionalidades do Lucene tornam o buscador de índices até bastante simples – a maior parte do trabalho está na indexação. Isso acontece porque o Lucene já traz o suporte a `RemoteSearchable`, uma forma simples de encapsular um índice aberto (um objeto `Searchable`) a um serviço JRMI. Objetos do tipo `RemoteSearchable`, como quaisquer objetos distribuídos, devem ainda ser registrados em algum serviço centralizado, de forma a serem localizados e utilizados por outras estações.

Dessa forma, este trabalho teve nesta área uma preocupação maior na forma de utilização destes objetos pelo *broker* e pelos nós. O Lucene em si possui duas abordagens de consultar múltiplos índices: o `MultiSearcher` e o `ParallelSearcher`. O primeiro realiza a consulta sequencialmente nos objetos `Searchable` conhecidos, unindo os resultados de cada consulta. Já o segundo cria uma nova *thread* para cada objeto `Searchable` que dispara as consultas; uma vez que todas as *threads* tenham completado, os resultados são unidos.

Imaginando que esperamos um sistema distribuído escalável, mesmo para um grande número de máquinas, o fato da abordagem de `ParallelSearcher` criar muitas *threads* para cada consulta nos parece um tanto ineficiente, dado o custo de criação de uma *thread* em um sistema sob uma carga significativa. Desenvolvemos então uma variação, o *ConcurrentBrokerSearcher*, que utiliza um *pool de threads* reaproveitáveis para essa tarefa. Os testes de desempenho para esta variação são realizados na seção 6.4.1.

Devido a proposta de utilização de múltiplos segmentos de índice por *index searcher* (os nós), a questão de utilização de `MultiSearcher` e `ParallelSearcher` também existe nesse contexto e experimentos comparativos serão realizados na seção 6.4.2.

Capítulo 6

Experimentos

6.1 Hardware e Software Básico

Os experimentos distribuíos realizados neste trabalho foram desenvolvidos no ambiente da empresa UpLexis[1] com hardware cedido pela mesma, consistindo em:

- Cliente: 1 Intel Core2Duo 6600 2.40GHz;
- Broker: 1 Intel Core2Duo 6600 2.40GHz;
- 8 Nós: 4 Intel Pentium D 3.00GHz e 4 Intel Core2Duo 6600 2.40GHz;
- Rede: Switch Gigabit Ethernet.

Todas as máquinas do *cluster* contém também 4GB de RAM DDR2-400, placas de rede gigabit ethernet e discos SATA 7200RPM de 1TB ou 750GB.

Para os testes realizados na seção 6.4.3 com índices não-distribuídos em uma única máquina de muitos recursos, se trata de um Dell PowerEdge 2950, com a seguinte configuração: Intel Xeon E5430 de 4 núcleos, 8 GB de memória RAM DDR2-667 e com os discos 3 SATA de 7200 RPM e capacidade 1.5TB e mais discos 2 SAS de 12000 RPM e capacidade de 450GB em RAID 0. As estruturas de dados utilizadas no trabalho (índices e *tablets*), estão nos discos SAS em RAID 0 (devido a maior velocidade).

Em termos de custos, o *cluster* inteiro estaria avaliado na casa dos R$10.000 e a máquina mais poderosa na casa dos R$20.000.

[1] http://www.uplexis.com

Todas as máquinas possuem instalados o sistema operacional Slackware Linux 64-bit 12.2.0, Java Virtual Machine 1.6.0 e Apache Lucene 2.3.2.

6.2 Domínio de Aplicação e Dados

O sistema de indexação e busca desenvolvido neste trabalho utiliza como *corpus* uma base de diários oficiais do estado de São Paulo, da União e de outros 13 estados brasileiros. O diário oficial é uma publicação diária dividida em cadernos (ou seções) que tem como função organizar e divulgar as leis, decisões administrativas, atos e resoluções legais das três esferas do governo – executivo, legislativo e judiciário –, além de um caderno para publicação de atas, balanços e editais para utilização das companhias de capital aberto. Em São Paulo, o diário oficial pode ser gratuitamente obtido no site da Imprensa Oficial[2].

Os diários oficiais, apesar de diários, são publicações enormes e em crescimento constante. Em 2008, o diário oficial de São Paulo era uma publicação com aproximadamente 6,5 mil páginas diárias; em 2010, já chega a 13 mil páginas. Somando diários de 14 estados do Brasil, mais os nacionais, a ClipDO[3] processa cerca de 60 mil páginas/dia, com picos de até 100 mil. Esses PDFs convertidos para texto pleno ocupam entre 1 e 2GB.

Apesar do conteúdo dos diários também ser de interesse da população em geral, são certamente os escritórios de advocacia seus leitores mais assíduos. Este interesse se deve a divulgação do estado atual de processos em andamento, que devem ser procurados manualmente no imenso diário – um trabalho geralmente dedicado a estagiários destes escritórios. Segundo a OAB, existem cerca de 150 mil advogados licenciados e ativos no estado de São Paulo e mais de meio milhão no Brasil.

Escritórios de advocacia podem também contratar o serviço de empresas especializadas em obter esse tipo de informação dos diários. Múltiplas empresas vendem este serviço com diferentes graus de precisão: enquanto algumas apenas terceirizam a busca manual, outras utilizam métodos computacionais básicos – como buscas por um conjunto de expressões regulares. Ao final do processamento dos cadernos, os clientes recebem por e-mail ou fax um relatório com as ocorrências de seu interesse no diário oficial daquele dia. Entretanto, nenhuma das duas alternativas permite muita flexibilidade: os assuntos/termos de interesse devem ser préviamente cadastrados e não é possível procurar em diários de dias anteriores.

[2] http://www.imprensaoficial.com.br/
[3] http://www.clipdo.com.br

Atualmente, a única forma disponível de realizar consultas em diários oficiais é por meio do próprio *website* da Imprensa Oficial, que utiliza o sistema de busca da Fast Search & Transfer (FAST), adquirida pela Microsoft em 2008. Além dos diários recentes, houve a digitalização de diversos diários antigos, de 1891 até o presente, em diferentes níveis de precisão e qualidade. Buscas por cadernos a partir de 1991 são tarifadas, pela taxa de R$12,00 por hora de uso. A quantidade de páginas indexadas por esse sistema está na casa das 9 milhões de páginas[4], com conteúdo disponível a partir de janeiro de 2003. O sistema possui um tempo de resposta alto para buscas, em especial para buscas por frase. Uma busca por "maria das graças aparecida leva cerca de 23 segundos para ser completada[5]. Outro grande problema do sistema é falta de capacidade em realizar buscas inexatas, um fator essencial para consultas no diário oficial devido a grande quantidade de erros existente. Estes erros são geralmente divididos em:

- Erros de digitação (extremamente comuns), que podem eliminar ou adicionar letras, ou mesmo utilizar letras erradas (como trocas de 'm' por 'n', 's' por 'z', dentre outros);
- Estado: se a fonte tem tamanho alterado, negrito, itálico... Um termo destacado tem maior peso.
- Erros de digitalização de conteúdo (erros de OCR – *Optical Character Recognition* – do software de reconhecimento de imagens) com efeito semelhante aos erros de digitação, mais comuns nos diários oficiais de São Paulo até meados de 2007 e ainda muito comuns nos diários de quase todo o país.

Estes erros tornam o domínio dos diários oficiais muito complexo de ser trabalhado com qualidade, tornando buscas exatas pouco eficazes por deixar de encontrar muito conteúdo relevante na base indexada. Não fosse o bastante, ainda se trata de um domínio que cresce diariamente em milhares de novos documentos.

Como cliente da UpLexis, a ClipDO cedeu sua base de diários oficiais processados para este trabalho.

6.3 Tablets

Para as Tablets, os dois testes mais importantes são aqueles para as duas operações mais significativas e frequentes: inserir mais dados e recuperar dados. A velocidade de operações

[4] fevereiro/2010 – 9.433.375 resultados para a busca pela letra 'a'
[5] busca realizada em fevereiro/2010

como *delete* e *update* não são fundamentais para o desempenho geral do sistema. Os 3 experimentos desta seção são *locais* (não-distribuídos), uma vez que deve-se comparar o desempenho das *tablets* contra outros sistemas (não necessariamente distribuídos).

Os 5 sistemas selecionados para os comparativos de desempenho contra as Tablets são:

- Lucene: algumas soluções, como o Apache Solr (descrito na seção 4.2.5), armazenam a coleção das estruturas criadas pela própria *engine* de busca – no caso, o Lucene;
- MySQL[6]: o SGBD mais utilizado no mundo, uma escolha óbvia para este experimento – versão 5.1.22 e tabelas do tipo MyISAM;
- PostgreSQL[7]: selecionado por ser considerado um SGBD mais robusto que o MySQL para grandes coleções – versão 8.3.6;
- SingleDir: cenário onde a coleção inteira está armazenada como arquivos num único diretório no sistema de arquivos ReiserFS[8] 3.6 (padronização da empresa dona do *cluster*);
- DirTree: variação do cenário acima, mas saturando menos o sistema de arquivos; a coleção é espalhada igualmente numa árvore de diretório em 3 níveis de altura, cada uma com 10 sub-diretórios;

A base de documentos utilizada para estes experimentos é totalmente artifical: 300 mil arquivos em texto pleno com o tamanho de 20KB, totalizando 6GB de informação.

Todos os testes realizados nessa seção foram executados na máquina *broker*.

6.3.1 Desempenho de Inserções

Neste experimento, medimos o tempo de inserção da coleção em cada um dos cenários. Na figura 6.1 e na tabela 6.1 temos os resultados com a média de 10 baterias do experimento.

O sistema de Tablets claramente tem uma operação de *insert* bastante eficiente. Seu desempenho supera a dos outros sistemas entre 38% (para o Lucene) e 128% (para o PostgreSQL).

[6] http://www.mysql.org
[7] http://www.postgresql.org
[8] http://ftp.kernel.org/pub/linux/utils/fs/reiserfs/

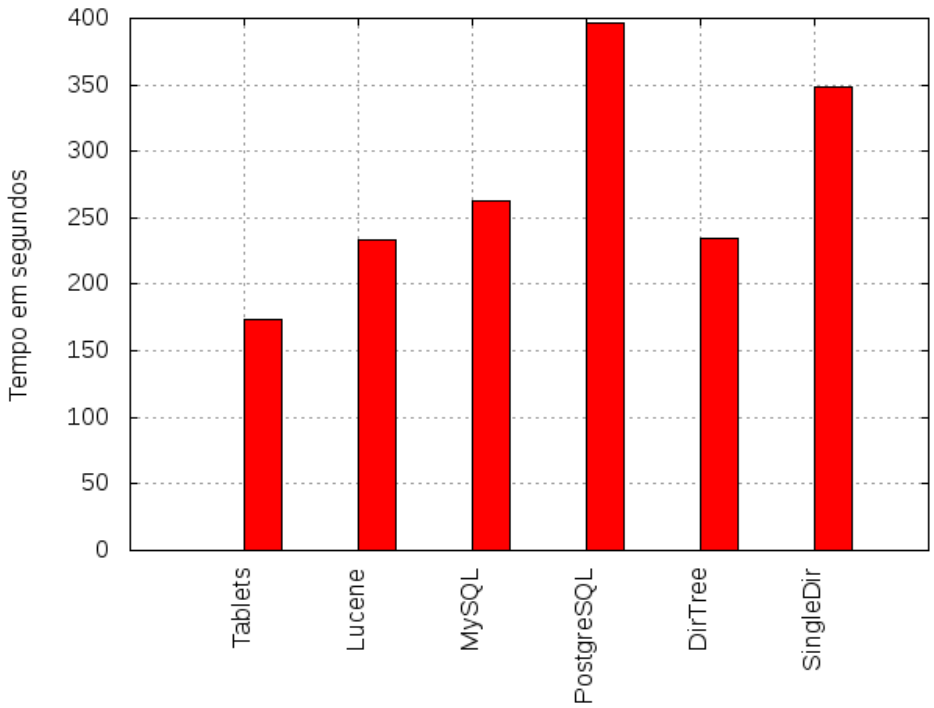

Figura 6.1: Experimento: inserção em tablets vs. outras soluções

Caso	Tablets	Lucene	MySQL	PostgreSQL	DirTree	SingleDir
Inserção	173.1	233.4	262.7	396.2	234.2	348.5
Busca	2.7	10.1	13.8	16.1	26.6	44.5
Busca Concorrente	3.6	16.2	17.9	18.4	33.9	55.1

Tabela 6.1: Dados tabelados comparativos entre soluções de recuperação de arquivos

6.3.2 Desempenho de Buscas

Neste experimento medimos o tempo necessário para cada solução recuperar uma listagem de 25 mil documentos armazenados no experimento anterior. Este experimento foi executado em 30 baterias, sendo cada uma delas formada por uma listagem diferente – as mesmas baterias foram executadas em todas soluções. Na figura 6.2 e na tabela 6.1 temos os resultados com a média das 30 baterias.

Antes da execução de cada bateria para uma dada solução, primeiramente foi realizada um 'aquecimento' (*warmup*), de forma a garantir que as estruturas utilizadas pelas soluções estivessem devidamente carregadas antes da marcação de tempo da bateria. O *warmup* é constituído por recuperar 5 documentos que não estão na listagem da bateria.

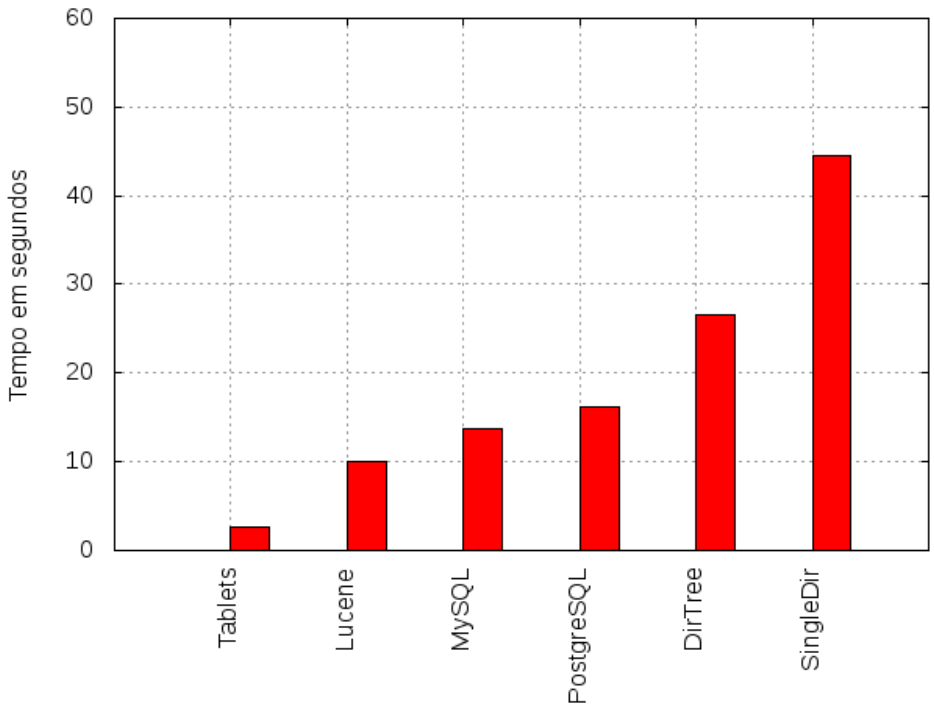

Figura 6.2: Experimento: busca em tablets vs. outras soluções

Se as Tablets demostravam um desempenho superior para operações de inserção, na recuperação essa vantagem é muito superior: a segunda melhor alternativa, o Lucene, é 3,7 vezes mais lento (2,7 x 10,1 segundos), enquanto que a pior opção, SingleDir, é 16,5 vezes mais lenta, comprovando que os conceitos e decisões da seção 5.2 são válidos.

Observando o comportamento das outras soluções pelo utilitário `strace`[9] é possível perceber múltiplas operações de entrada/saída para recuperar um único documento. Em Tablets, apenas uma operação é necessária, já que todos os arquivos de *tablets* (91 neste experimento) estão previamente abertos e as posições de disco já armazenadas em memória. Nas máquinas de busca tradicionais, para cada busca em índices realizada, é necessário recuperar *dez* documentos para a montagem dos *snippets*, aumentando ainda mais o impacto do desempenho desta atividade no tempo total de uma busca para o usuário.

6.3.3 Desempenho de Buscas Concorrentes

Após os experimentos de buscas sequenciais, é justo questionar se as Tablets também se comportam bem sob acesso concorrente, uma situação padrão em máquinas de busca.

[9] http://sourceforge.net/projects/strace/

Neste experimento, vamos repetir o experimento anterior (30 baterias de 25 mil consultas com *warmup*), com a diferença que dessa vez temos 4 *threads* concorrentes que estressam os sistemas. Na figura 6.3 e na tabela 6.1 temos os resultados com a média das 30 baterias.

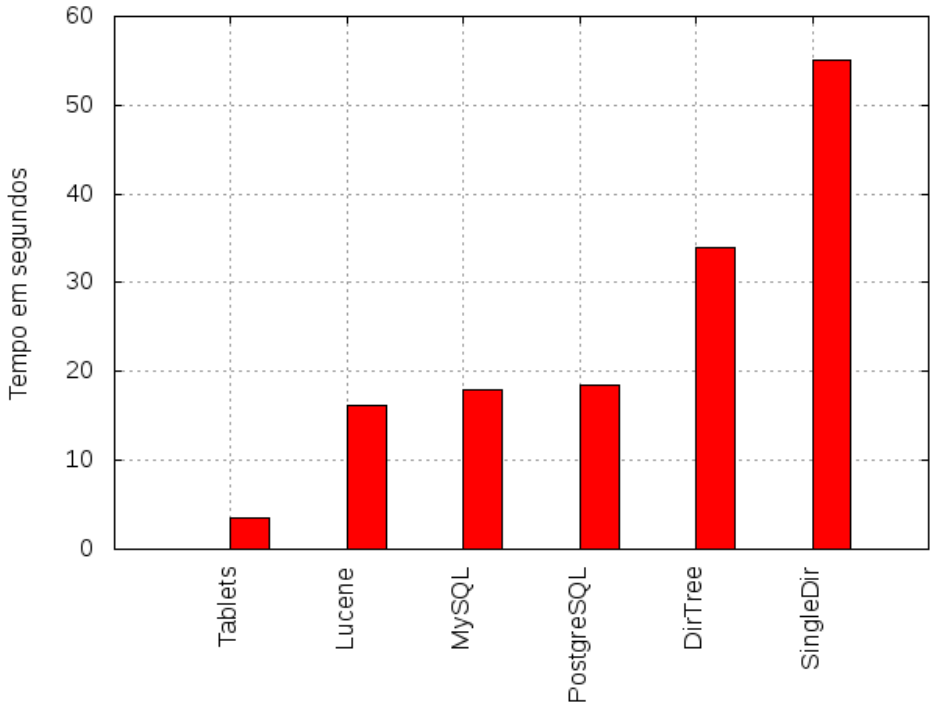

Figura 6.3: Experimento: busca concorrente em tablets vs. outras soluções

Neste cenário podemos notar que:

- O acesso concorrente piora o desempenho de todas as soluções;
- As Tablets continuam sendo a melhor solução;
- Os tempos do Lucene, MySQL e PostgreSQL se aproximaram bastante.

6.4 Buscas

Ao contrário da seção anterior que testava o subsistema de Tablets isoladamente, nos experimentos desta seção temos o sistema por completo disponível e sendo avaliado, primeiramente nas suas partes (*broker* e nós) e então como um todo. Nos experimentos dessa seção foi utilizado um *corpus* de 10,9 milhões de documentos, totalizando 97GB de dados. Esses documentos são as páginas de diários oficiais processados e cedidos pela ClipDO, como

explicado na seção 6.2.

Desenvolvemos uma pequena aplicação para *benchmarks* que dispara consultas XML-RPC para o *broker* (como descrito na seção 5.3.3, esta é a forma do sistema receber consultas externas) que será utilizada em todos os cenários a seguir. Essa aplicação é parametrizável em variáveis como o número de *threads* consultando o sistema, a URL para enviar consultas e a base de termos a serem consultados, dentre outros.

As consultas a serem realizadas são montadas sorteando 2 termos da listagem passada e uma operação (*AND, OR, NOT* ou frase[10]). Para a listagem de termos a serem sorteados para as consultas utilizamos os 51 termos mais frequentes da coleção, como 'justiça', 'processo' e 'maria', de tal forma que as consultas sejam um pouco trabalhosas ao sistema.

Os experimentos foram realizados com uma verificação de corretude nas consultas, por meio do registro da quantidade total de resultados que cada consulta obteve. Como todos os cenários foram executados com a mesma base e consultas, discrepâncias na quantidade de resultados seriam um sinal de problemas. Eles foram disparados da máquina *cliente* e executados em 30 baterias de 5 minutos de duração, com 10 consultas de *warmup*. Ressaltamos que o cliente e o *broker* estão localizados na mesma rede local *gigabit ethernet*, de forma que a baixa latência mantém initerruptas consultas ao sistema mesmo para um baixo número de *threads*.

6.4.1 Desempenho do Broker

Neste experimento, avaliamos o desempenho do sistema alternando a forma do *broker* realizar as consultas nos nós do sistema.

- MultS – *broker* utilizando o `MultiSearcher` (incluso no Lucene): repassa a busca nó a nó, coletando os resultados e os unindo ao final;
- ParS – *broker* utilizando o `ParallelSearcher` (incluso no Lucene): para cada busca, cria uma *thread* para cada nó; quando todas as *threads* de uma consulta tiverem terminado, junta os resultados;
- CBS – *broker* utilizando o `ConcurrentBrokerSearcher` (descrito na seção 5.3.3), uma variação otimizada do `ParallelSearcher` utilizando um *pool de threads*, desenvolvido neste trabalho;

[10] Exigir que os termos apareçam no texto em posições consecutivas

Para este experimento, executamos 2 cenários, cujos resultados estão na tabela 6.2, variando apenas a quantidade de *threads* simultâneas no aplicativo de *benchmark*. A figura 6.4 mostra os resultados do primeiro cenário deste experimento, utilizando 5 *threads* simultâneas, e a figura 6.5 mostra o 2º cenário, para 15 *threads*.

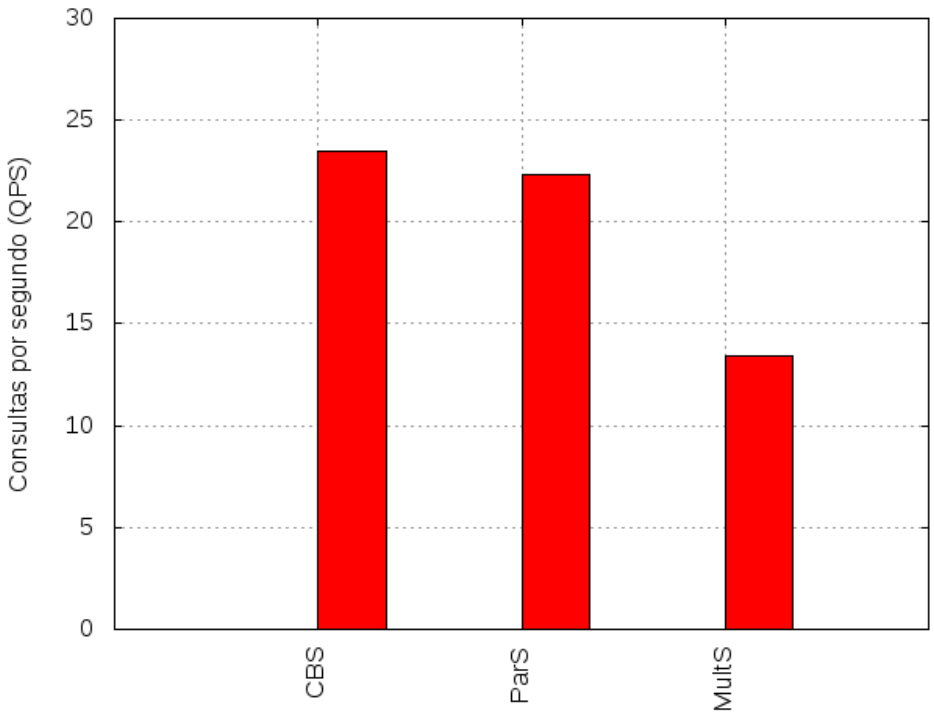

Figura 6.4: Experimento: análise do broker, 5 threads

Do primeiro para o segundo cenário podemos notar que o sistema demonstra uma grande saturação, reduzindo consideravelmente a vazão do sistema (QPS – *queries per second*). Entretanto, em ambos os cenários, a abordagem de disparar as consultas aos nós simultâneamente (ParS e CBS) é superior ao disparo sequêncial. Essa abordagem se mostra muito superior antes da saturação, sendo até 75% mais rápida. Após a saturação, a abordagem se torna pouco mais eficiente, tendo desempenho 10,6% melhor.

Threads	Estado	CBS	ParS	MultS
5	Não-Saturado	23.5	22.3	13.4
15	Saturado	11.4	10.7	10.3

Tabela 6.2: Dados tabelados comparativos entre estratégias para o *broker*

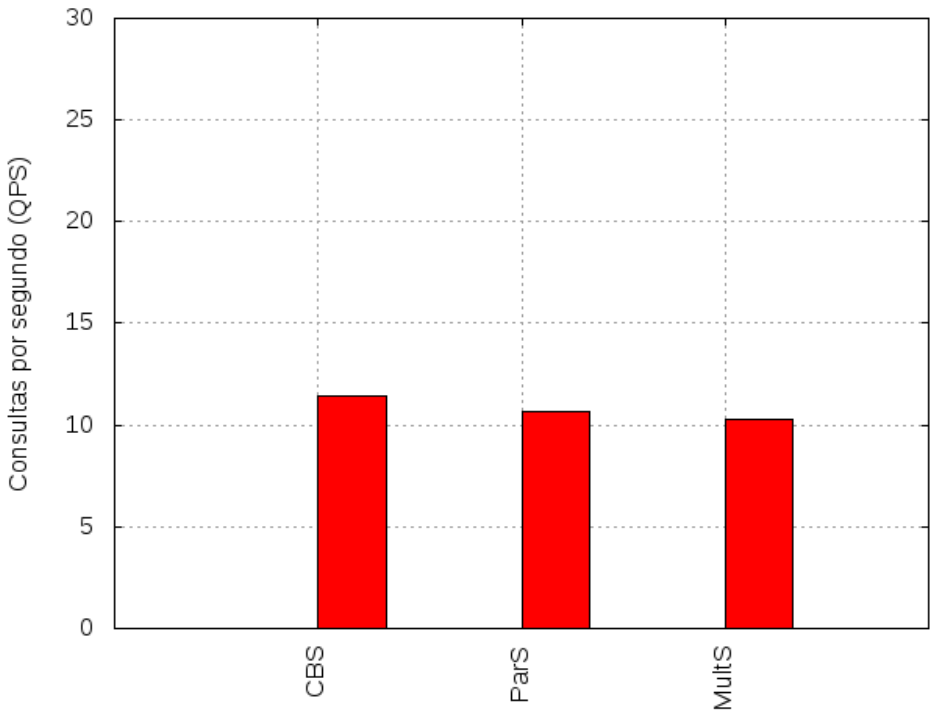

Figura 6.5: Experimento: análise do broker, 15 threads

Podemos notar também que nossa abordagem de CBS, utilizando o `ConcurrentBrokerSearcher`, se mostra ligeiramente superior ao ParS em ambos cenários (5,3% e 6,5% mais eficiente, respectivamente), de tal forma que o custo de criação de *threads* realmente existe e causa um certo impacto no desempenho geral do sistema.

6.4.2 Desempenho nos Nós: Índices Segmentados

Neste experimento, avaliaremos o desempenho dos nós (*index searchers*) do *cluster* utilizando um índice único e índices segmentados, consultados via `MultiSearcher` e `ParallelSearcher`. Novamente, realizamos o experimento nos cenários de 5 e 15 *threads* simultâneas para avaliar o sistema saturado e não-saturado. Experimentamos também a partição em 3 e 5 segmentos de índice. O resultados destes experimentos estão nas figuras 6.6 e 6.7.

Threads	Estado	Single	ParS-3	ParS-5	MultS-3	MultS-5
5	Não-Saturado	22.3	20.2	19.1	21.1	19.8
15	Saturado	11.4	9.4	9.1	10.5	10.0

Tabela 6.3: Dados tabelados comparativos entre estratégias de segmentação de índices

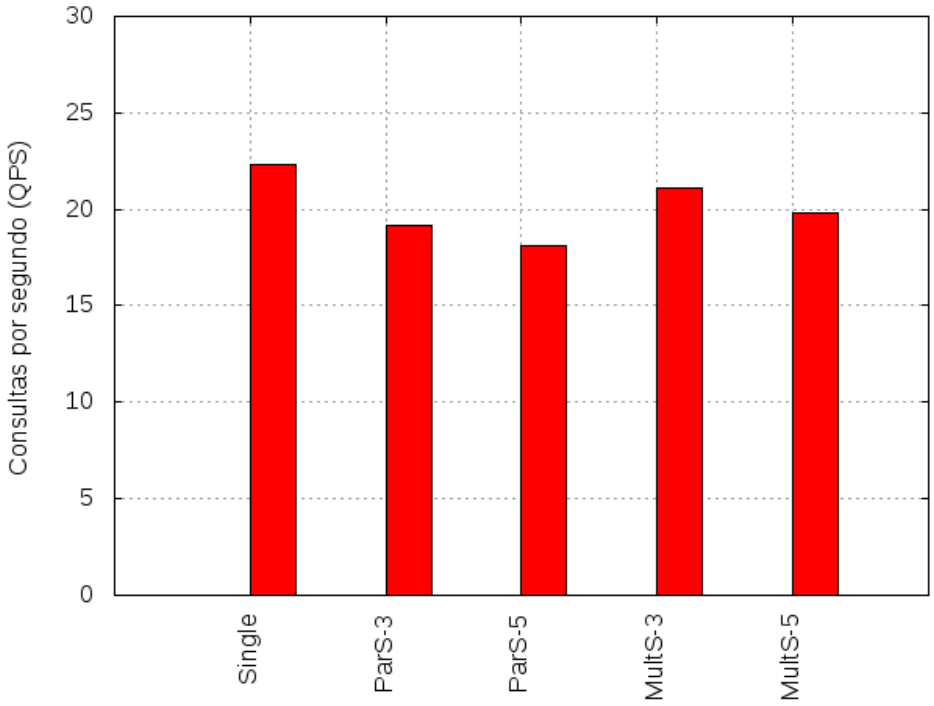

Figura 6.6: Experimento: análise de nós, 5 threads

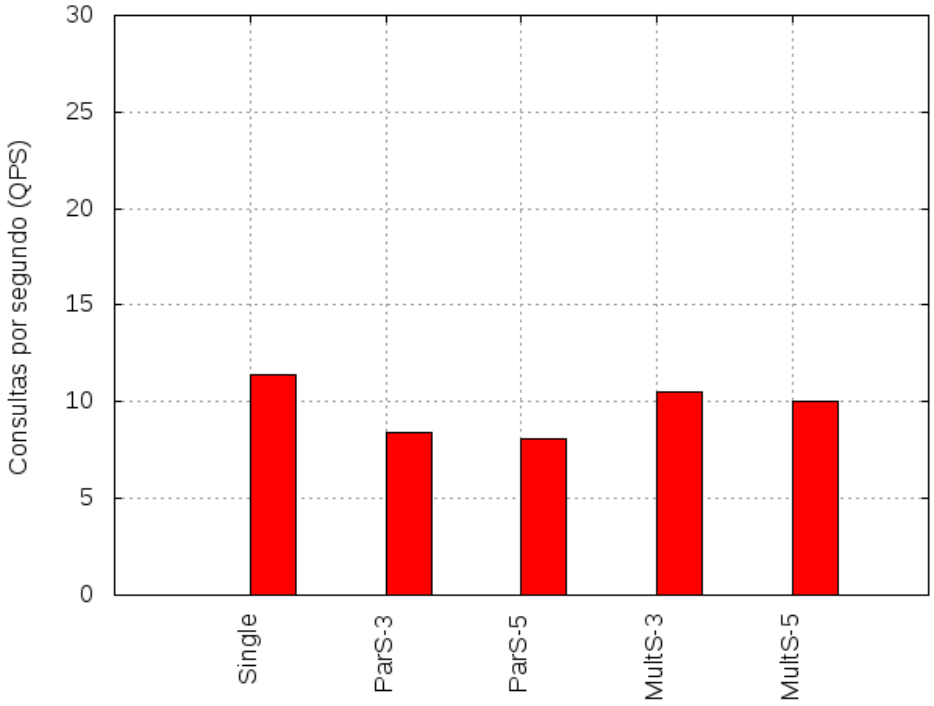

Figura 6.7: Experimento: análise de nós, 15 threads

Como já esperado, para um dado nó, o desempenho de um índice único é mais eficiente que a utilização de múltiplos índices – as estruturas de dados se tornam mais otimizadas e não há necessidade de unificar resultados após a busca. Neste experimento, o impacto da segmentação foi inferior ao esperado, além de notarmos a tendência de que em um sistema já saturado, a quantidade de segmentos têm um impacto maior do que em um não saturado.

- Não-Saturado: a perda de desempenho (para MultS) foi de 5,4% para 3 segmentos e de 11,3% para 5 segmentos;
- Em saturação: a perda de desempenho (para MultS) foi de 7,9% para 3 segmentos e de 12,3% para 5 segmentos.

Acreditamos que o baixo impacto da utilização de segmentação seria compensada pelas vantagens da proposta de utilização de segmentação e do Hadoop para alcançar um bom nível de escalabilidade e tolerância a falhas, propostos na seção 5.3.1.

Apesar de ser possível, o `ConcurrentBrokerSearcher` não foi avaliado neste experimento, uma vez que o `ParallelSearcher` não se comportou bem. Esse desempenho inferior comparado ao `MultiSearcher` (o contrário do que ocorre no *broker*) acontece devido à disputa das *threads* pela utilização do disco, um problema bem conhecido na programação concorrente. Acreditamos que em um cenário no qual cada segmento de índice estivesse em um disco rígido diferente, o comportamento do `ParallelSearcher` (e consequentemente do `ConcurrentBrokerSearcher`) seria mais uma vez superior.

6.4.3 Desempenho do Sistema Completo

Neste experimento, nosso objetivo é analisar o desempenho do sistema como um todo, comparando-o em diferentes situações e contra o Apache Solr (vide seção 4.2.5), cujos resultados estão na tabela 6.4:

- L+T: esta solução se trata do sistema completo desenvolvido neste trabalho, com índices distribuídos e Tablets, rodando no *cluster* de 8 nós já descrito;
- Solr: a utilização do Apache Solr 1.3 (na configuração padrão) para resolver o mesmo problema, rodando no *cluster* de 8 nós já descrito;
- Local1: a solução deste trabalho rodando de forma não-distribuída, na máquina *broker*;

- Local2: a solução deste trabalho rodando de forma não-distribuída, na máquina Dell PowerEdge.

L+T	Solr	Local1	Local2
11.4	8.3	2.28	7.6

Tabela 6.4: Dados tabelados comparativos entre desempenho geral dos sistemas

Realizamos o teste com o grande *corpus* de diários oficiais e a base de consultas elaborada no cenário de 15 *threads* simultâneas, visto nos experimentos anteriores como uma situação estressante ao sistema. O resultado deste experimento pode ser visto na figura 6.8.

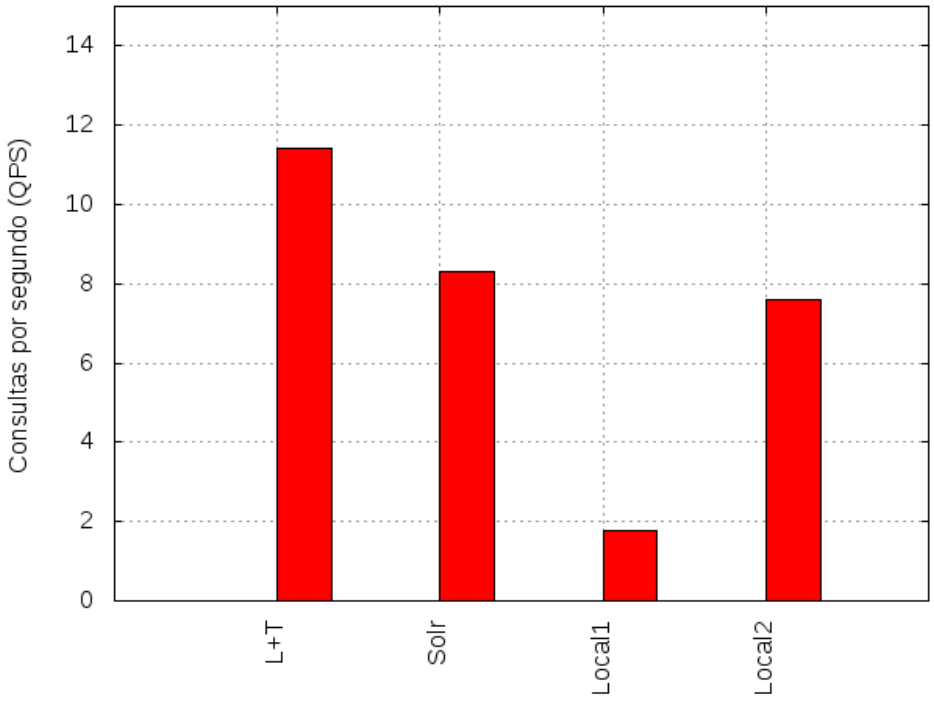

Figura 6.8: Experimento: desempenho geral dos sistemas

Podemos verificar que:

- *Speedup* e eficiência [7]: analisando as métricas básicas de sistemas paralelos e distribuídos, a solução deste trabalho num *cluster* de 8 nós comparado a `Local1` teve *speedup* de 6,4 e eficiência de 80%, bons números e relativamente próximos ao (desejado, mas improvável) crescimento linear;

- A solução deste trabalho mostrou uma vazão 37,4% superior a do `Solr`, um valor bem expressivo para o mesmo hardware; nos fatores envolvidos aqui, podemos citar a comunicação mais eficiente (JRMI x SOAP) e a recuperação de documentos mais eficiente (Tablets x Lucene); entretanto, não foi possível isolar e avaliar somente o desempenho de índices distribuídos das duas soluções;

- O cenário deste trabalho `L+T` apresentou uma vazão 49,5% superior ao cenário `Local2`; o custo de hardware do cenário `Local2` é mais que o dobro que o *cluster* utilizado;

6.4.4 Escalabilidade e Tolerância a Falhas

Neste experimento, visamos analisar a validade da utilização de segmentos para se obter escalabilidade e tolerância a falhas de forma ágil, equilibrada e sem a necessidade de reindexar toda a coleção. Assim, vamos analisar dois cenários de indexação, com e sem segmentos, alterando a quantidade de máquinas disponíveis no *cluster*, para mais e para menos. Dessa forma, pode-se verificar a variação de desempenho do sistema em caso de falhas e de adição de novas máquinas.

Dado o *cluster* de 8 máquinas disponível para a realização dos testes, optamos por utilizar como referência uma configuração com 6 máquinas, variando a quantidade entre -2 e +2 máquinas. A quantidade de segmentos utilizada na indexação foi de *N-1*, onde N é o número de máquinas utilizadas na indexação. Escolhemos este número por se tratar de uma boa solução para a situação mais provável das analisadas: a queda de uma máquina do *cluster* (geralmente por problemas de hardware). Quando uma máquina cair, os segmentos pelos quais ela era responsável deverão ser distribuídos pelas outras máquinas: nesse caso, a utilização de N-1 segmentos torna ótima a distribuição entre N-1 máquinas.

Em cada uma das 6 máquinas, utilizamos uma coleção de 5 milhões de documentos, totalizando 30 milhões de documentos para o sistema inteiro. Cada máquina gerou um índice sem segmentação e outro índice formado por 5 segmentos (totalizando 30 segmentos no sistema inteiro). Assim, a divisão dos índices nos 5 casos que foram analisados é feita da seguinte forma:

Caso 0 Cluster com 4 máquinas. Para segmentação, duas máquinas com 8 segmentos e outras duas com 7 segmentos. Para o teste sem segmentação, duas máquinas estão com 2 índices e outras duas com 1 índice;

Caso 1 Cluster com 5 máquinas. Para segmentação, cada máquina é responsável por 6 segmentos. Para o teste sem segmentação, quatro máquinas estão com 1 índice e uma

com 2 índices;

Caso 2 Cluster com 6 máquinas. O caso base. Para segmentação, cada máquina é responsável por 5 segmentos. Para o teste sem segmentação, cada máquina contém 1 índice;

Caso 3 Cluster com 7 máquinas. Para segmentação, temos cinco máquinas com 4 segmentos e duas máquinas com 5. Para o teste sem segmentação, é idêntico ao caso 2, mais uma máquina ociosa;

Caso 4 Cluster com 8 máquinas. Para segmentação, temos seis máquinas com 4 segmentos e duas máquinas com 3. Para o teste sem segmentação, é idêntico ao caso 2, mais duas máquinas ociosas;

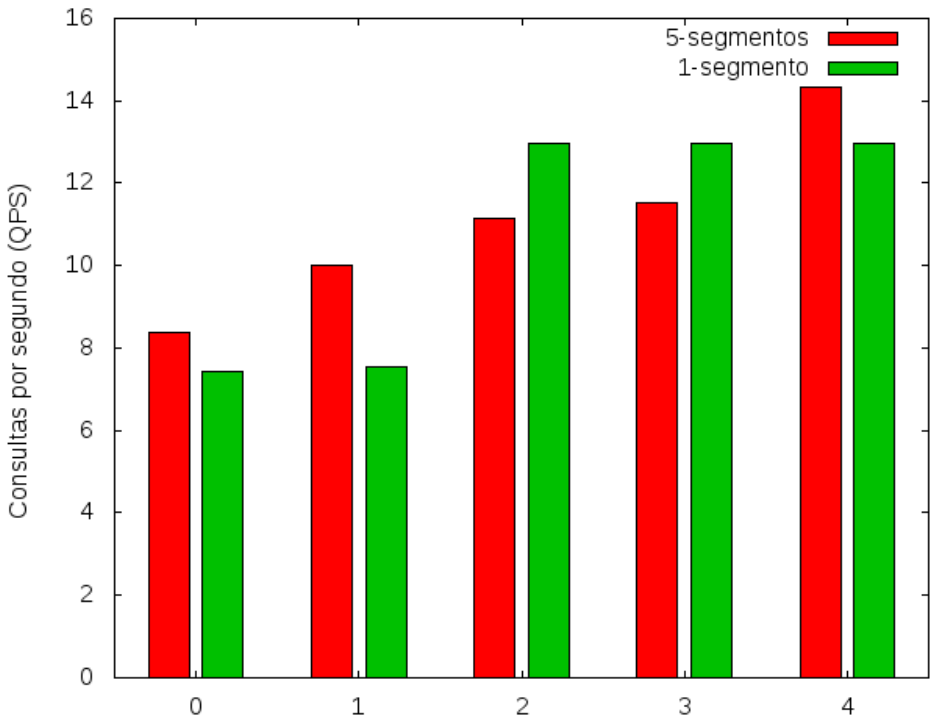

Figura 6.9: Experimento: desempenho de índice segmentado e tradicional variando tamanho do *cluster*

Nos testes realizados, cujos resultados estão na figura 6.9 e tabela 6.5, notamos que:

- O índice não-segmentado tem a mesma vazão quando o *cluster* aumenta, uma vez que não há o que se repassar para as novas máquinas (haverá somente em caso de reindexação);

Caso	Cluster	1-seg QPS	5-seg QPS	Ganho 5-seg
0	4	7.44	8.38	+12,6%
1	5	7.56	10.02	+32,5%
2	6	12.98	11.13	-14,2%
3	7	12.98	11.54	-11,1%
4	8	12.98	14.33	+10,4%

Tabela 6.5: Dados tabelados comparativos de índice segmentado e tradicional variando tamanho do *cluster*

- Quando se aumenta o *cluster* em apenas 1 máquina, a melhoria de desempenho do sistema baseado em segmentos é muito pequena (3,6%), uma vez que ainda temos estações com a mesma quantidade de segmentos do caso base – se tornam o gargalo do sistema. Para melhorar a vazão, precisamos que todas as máquinas reduzam a quantidade de segmentos. Para que isso aconteça com o acréscimo de apenas 1 máquina, a quantidade de segmentos por máquina precisa ser muito grande, impactando o desempenho do sistema. Vale ressaltar que, se houver reindexação, haverá aumento de desempenho para este caso;

- Quando o *cluster* aumenta em 2 máquinas, a abordagem de N-1 segmentos já reduz a quantidade de segmentos de 5 do caso base para um máximo de 4 por máquina , acarretando no aumento de 10,4% na vazão. Quando for realizada uma reindexação o desempenho aumentará mais um pouco, uma vez que todas as máquinas estarão com mesma carga;

- Quando o *cluster* reduz em 1 máquina, nossa estratégia de N-1 segmentos representa o caso ótimo de divisão, com todas as máquinas com mesma carga (N segmentos). Enquanto isso, a solução padrão não-segmentada gera um grande gargalo em uma só máquina que tem sua carga dobrada – o que acarreta em nossa estratégia apresentar uma vazão 32% superior. Por este se tratar do caso mais comum dos analisados, consideramos o teste mais relevante;

- Quando o *cluster* reduz em 2 máquinas, temos o cenário sem-segmentação muito semelhante ao caso anterior, mas agora são 2 máquinas com o dobro de carga – o que não muda o desempenho. No caso de segmentação, temos máquinas já com uma grande quantidade de segmentos (até oito) e carga (60% maior que o caso base). Na comparação destas duas situações, nossa solução também mostra vazão superior, em 12,6%.

6.5 Resumo dos Experimentos

- Desempenho de Tablets – comparamos a solução desenvolvida neste trabalho com soluções *open source* comumente utilizadas, com nossos testes mostrando um desempenho superior estas soluções. Nossa solução, comparada ao Lucene – a solução de melhor desempenho das comparadas – apresenta desempenho superior: 38% mais rápido nas inserções e 274% nas buscas;

- Desempenho do Broker – nossos testes revelam que, no *broker*, a utilização de buscas em paralelo (via `ParallelSearcher`) são até 75% mais eficientes que buscas sequenciais (via `MultiSearcher`). Na nossa solução, o `ConcurrentBrokerSearcher` reaproveita as *threads* do `ParallelSearcher` e alcança um desempenho até 6,5% superior a este;

- Desempenho de Índice Segmentado – nossos testes compararam o desempenho de um índice comum contra índices segmentados, nos quais, como esperado, quanto menos segmentos, maior o desempenho. Além disso, a estratégia de buscar nos índices segmentados sequencialmente (via `MultiSearcher`) é mais eficiente que a busca em paralelo (via `ParallelSearcher`), devido aos limites do acesso paralelo ao disco;

- Desempenho do Sistema Completo – comparamos nossa solução contra a solução distribuída mais utilizada e baseada em Lucene, o Solr, e contra dois cenários não-distribuídos, uma com o hardware mais caro que o *cluster* inteiro e outra com apenas uma máquina do *cluster*. Nossos testes mostraram que nossa solução tem uma eficiência de 80% para um *cluster* de 8 nós e que tem desempenho 37,4% superior ao Solr (para o mesmo *cluster*) e 49,5% superior ao cenário não-distribuído e de hardware caro;

- Desempenho de Segmentos para Tolerância a Falhas – comparamos a utilização de N-1 segmentos (onde N é a quantidade de nós do *cluster*) por nó com a abordagem tradicional (sem segmentos), enquanto variamos a quantidade de nós no *cluster*. Utilizando um caso base do *cluster* com 6 nós, avaliamos a divisão dos mesmos índices gerados (sem reindexação) em cenários de 4 e 5 nós no *cluster*. Nossa abordagem de segmentação se mostrou superior no caso da redução de máquinas, 12,6% reduzindo duas máquinas e 32% reduzindo apenas uma máquina (o caso mais comum).

- Desempenho de Segmentos para Escalabilidade – utilizando a mesma indexação do caso anterior, dessa vez variamos para cima a quantidade de máquinas no *cluster*, com 7 e 8 máquinas. Nossa abordagem de segmentação para o acréscimo de uma máquina não se mostrou eficaz, uma vez que os nós, no pior caso, mantiveram a mesma quantidade

de segmentos (e carga) do caso base. Entretanto, para o aumento de duas máquinas, conseguimos um desempenho de 10,4% ao caso base, sem necessidade de reindexação.

Capítulo 7

Conclusões

7.1 Considerações Finais

Neste trabalho, primeiramente fizemos um levantamento de diferentes arquiteturas de sistemas distribuídos para indexação e busca em sistemas de Recuperação de Informação, avaliando vantagens e desvantagens destas arquiteturas. Dentre as arquiteturas estudadas estão as máquinas de busca do Google, YaCy e Grid Lucene. A seguir, guiados por trabalhos do Google, Apache Solr e TodoBR, realizamos estudos sobre as estratégias de distribuição das estruturas de dados importantes ao sistema, analisando o impacto e a importância de suas estruturas e subsistemas.

Com este embasamento teórico/experimental, desenvolvemos uma implementação própria de uma máquina de busca distribuída, incorporando diferentes decisões de projeto de diversos trabalhos. Além disso, iniciamos o desenvolvimento de abordagens que tentam aprimorar o desempenho dessas soluções, como o `ConcurrentBrokerSearcher` – a ser disponibilizado à comunidade Lucene como um *patch* –, ou aspectos de escalabilidade e tolerância a falhas, como a segmentação de índices nos nós – uma decisão de projeto que não vimos ser descrita e analisada em outros trabalhos estudados.

Nossos experimentos confirmaram as boas decisões de projeto e implementação, mostrando que Tablets e `ConcurrentBrokerSearcher` são superiores às opções disponíveis para utilização, alcançado boas melhorias de desempenho. Para os segmentos de índice, consideramos válidos os experimentos para se avaliar esta alternativa. A perda de desempenho devido a utilização de segmentos é bastante aceitável, considerando-se as vantagens e os ganhos de desempenho pela utilização de Tablets e do `ConcurrentBrokerSearcher`. Quando acontece uma variação da quantidade de máquinas, mas não uma nova indexação, nossa abordagem de segmentação obteve desempenho superior em tanto reduzindo quanto aumentando a quan-

tidade de máquinas disponíveis, dividindo a carga de forma eficiente. A velocidade de se redistribuir os segmentos nesse cenário sem nova indexação, por meio da utilização de um sistema de arquivo distribuído, adiciona um grau de escalabilidade e tolerância a falhas ao sistema.

Por fim, analisando o sistema como um todo, nossa solução apresentou um desempenho muito superior (vazão 49,5% superior) a uma solução não-distribuída utilizando um *hardware* mais que 2 vezes mais caro que o *cluster* inteiro. Comparada ao largamente utilizado Apache Solr, esta implementação, baseada no Lucene e Tablets, também se mostrou como sendo a mais eficiente, com menores tempos de resposta e maior vazão em todos os testes que pudemos realizar. A vazão, por exemplo, foi 37,4% maior em nosso sistema, se comparado com a solução equivalente do Apache Solr 1.3 em sua configuração padrão. Entretanto, devemos ressaltar que o Apache Solr apresenta também grandes virtudes frente a este trabalho, como um sistema genérico pronto para *deploy*, com ferramentas de configuração e administração web muito eficientes, além de uma série de *plugins* que extendem suas funcionalidades.

Nossas contribuições principais neste trabalho foram:

- Desenvolvimento de um *patch* para o *Lucene* – uma classe que é capaz de realizar buscas de forma mais eficiente que o `ParallelSearcher`, a qual será submetida à comunidade na forma de um *patch*;
- Tablets – um estudo e implementação de um sistema de alta performance para o armazenamento e recuperação dos textos originais da coleção, um setor de pouca atenção na literatura;
- Segmentação de Índices – uma abordagem de indexação que, ao custo de um desempenho ligeiramente inferior, facilitará aspectos de balanceamento de carga e tolerância a falhas para sistemas de busca distribuídos;
- Desenvolvimento do Buscador DO – uma eficiente ferramenta de busca em Diários Oficiais de diversos estados do Brasil, superior àquela disponível no *site* da Imprensa Oficial de São Paulo.

7.2 Sugestões para Pesquisas Futuras

- Experimentos – experimentos mais extensivos do ponto de vista de sistemas distribuídos podem ser realizados, como a análise do desempenho mediante o crescimento da quantidade de máquinas: 1, 2, 4, 8 e até *clusters* maiores que este utilizado. Nestes

experimentos, poderia-se avaliar as tendências de *speedup*/eficiência, vazão (consultas resolvidas por segundo) e o tempo de resposta médio;

- Isolamento dos tempos de consulta Solr – em nossos experimentos fomos incapazes de estabelecer exatamente qual o peso de cada fator que tornou este sistema superior ao Apache Solr. Um estudo nesse sentido seria muito útil para análise de impacto de soluções;

- Fusão com Solr – a sinergia das virtudes da arquitetura deste trabalho, aliado com as caraterísticas 'comerciais' e de facilidade de uso do Solr certamente fariam um híbrido superior; o fato de ambos se basearem em Lucene colabora fortemente para o sucesso desta fusão;

- Índices Globais – encontrar uma forma de construir índices globais utilizando o Lucene que seja o mínimo possível agressivo com a estrutura interna da biblioteca;

- Escalabilidade e Tolerância a Falhas – com a ideia de segmentação, viria uma grande integração com o Hadoop; o *broker* passaria a ser responsável por conhecer o estado exato das máquinas e decidir momentos de redistribuição das estruturas e seus responsáveis. Diversos algoritmos de escalonamento, *heartbeat* e balanceamento podem ser analisados aqui;

Apêndice A

Buscas Inexatas

A capacidade de realizar buscas inexatas é de extrema importância para os usuários de máquinas de busca que lidam com um domínio com uma coleção tão suscetível à erros ortográficos e de digitação como é o de Diários Oficiais (como descrito na seção 6.2). Palavras como 'Bradesco' possuem mais de 10 variações (erradas) que ocorrem muito frequentemente, como 'Brasdesco', 'Bredesco' e 'Bradescos'. Isso implica que grande parte das buscas realizadas não retornam diversas ocorrências sobre aqueles termos desejados pelo usuário.

Embora o Lucene seja uma biblioteca de indexação e busca extremamente complexa e com muitos operadores, nenhum deles é capaz de atingir o resultado esperado para esse tipo de busca. Dessa forma, criamos um operador de *aproximação* que pode ser utilizado pelos usuários – ele transforma uma busca por 'Bradesco' em uma busca por 'Bradesco AND Bradescos AND Bredesco AND Brasdesco ...'. Para isto, utilizamos a estrutura de dados *trie* e o algoritmo de *distância de Levenshtein* [26].

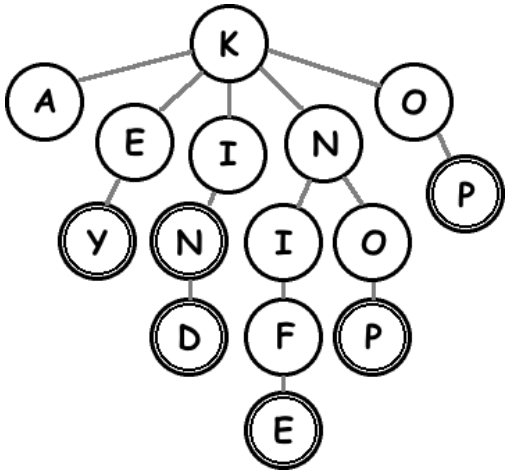

Figura A.1: Trie gerada com uma pequena coleção de palavras

,

Em nosso fluxo de indexação a última fase recupera o vocabulário do índice gerado pelo Lucene e o transforma em uma *trie* – a *trie* gerada após a indexação da nossa coleção de 11 milhões de páginas de Diários Oficiais ocupa cerca de 300MB. A grande vantagem da utilização de *tries* é a velocidade de busca por uma chave: tempo O(m), onde m é o tamanho da chave, contra O(log(n)) para buscas em uma árvore binária, onde n é a altura da árvore. Uma *trie* gerada com as palavras 'key', 'kin', 'kind', 'knife', 'knop' e 'kop' se assemelha a representada na figura A.1.

Uma vez com a *trie* gerada com o vocabulário do índice, utilizamos o algoritmo de distância de Levenshtein para localizar os termos que tenham *distância de edição* 1 com o termo buscado. Para uma distância de edição 1, pode haver apenas 1 letra adicionada, removida ou substituída (apesar de ser possível dar pesos diferentes para cada caso). Para a palavra 'Bradesco', isso significa, respectivamente, termos como 'Brasdesco', 'Bradesc' e 'Bredesco'. A figura A.2 mostra o cálculo da distância de edição entre duas palavras.

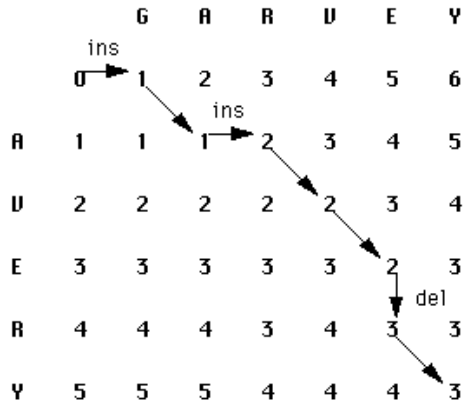

Figura A.2: Distância de Levenshtein entre os termos 'Garvey' e 'Avery'

A abordagem utilizada aqui é extremamente bem sucedida em conseguir encontrar resultados que, de outra forma, seriam perdidos pelos usuários, tornando nossa máquina de busca muito eficiente mesmo em um domínio complicado como o de Diários Oficiais. Além disso, nosso operador de aproximação é plenamente compatível com outros operadores do Lucene – assim, uma busca por frase (posições dos termos exatamente consecutivas) e aproximação por 'Banco Bradesco' retornaria documentos com ocorrências de 'Banc Brasdeco'. Por fim, a utilização de *tries* em memória RAM torna sua velocidade ainda mais expressiva e de baixo impacto para o buscador.

Por outro lado, muitas vezes a expansão de um termo para todos aqueles com distância de edição 1 ocasiona na aparição de termos indesejados – como 'Mario' quando se buscava por 'Maria'. Adicionalmente, algumas vezes os termos desejados podem estar em distância de edição 2 do termo original, de forma que mesmo com nosso sistema não seriam encontrados. Dessa forma, estamos estudando a atribuição de pesos para os termos encontrados na *trie*. Os fatores que estão sendo analisados para a atribuição de pesos são a frequência de aparição no índice, distância de edição do termo original e a frequência de co-ocorrência com o termo original.

Apêndice B

Compressão

A *compressão de dados* é uma técnica básica muito utilizada em diversos sistemas cujos principais objetivos são reduzir o tamanho de uma grande quantidade de dados e minimizar a duração de operações de entrada/saída, justamente pelo menor tamanho do bloco de dados.

No escopo dos sistemas de busca e indexação, a compressão pode ser muito útil para agilizar o carregamento de dados do disco para a memória (listas invertidas ou documentos) ou então, caso o segmento de índice seja grande demais para a memória principal, conseguir que a quantidade de listas invertidas carregadas em memória seja maior que o normal.

Como armazenamento em disco é um recurso computacionalmente barato, o principal objetivo aqui é a redução na duração das operações de entrada/saída. Porém, para a compressão ser útil dessa forma o tempo de carregamento do bloco compactado e mais a sua descompressão deve ser mais rápido do que o carregamento do dado sem nenhuma compactação. Dessa forma, os métodos leves com descompressão extremamente veloz são mais relevantes ao problema do que aqueles que obtém grandes taxas de compressão, mas que exigem muito esforço de processamento para descompressão.

B.1 Manipulação manual de bits

Esses métodos são reduções de utilização de bits determinas pelo próprio programador, utilizando menos bits que o normal em campos onde normalmente nunca se chegará em valores tão grandes. Por exemplo, os inteiros (sem sinal) possuem 32 bits (4 bytes), com valores variando entre 0 e mais de 4 bilhões (4.294.967.295). Os campos numéricos, como o identificador do termo, o identificador da coleção e a frequência no documento muito dificilmente alcançarão tal patamar.

No exemplo abaixo (figura B.1), foram considerados 3 bytes (16 milhões de possíveis valores) para os identificadores de documento (o que pode ser pouco para alguns sistemas) e de termo, e apenas 1 byte para o contador de frequência do termo no documento. Assim, uma estrutura que descompactada teria 40 bytes, compactada dessa forma teria apenas 18.

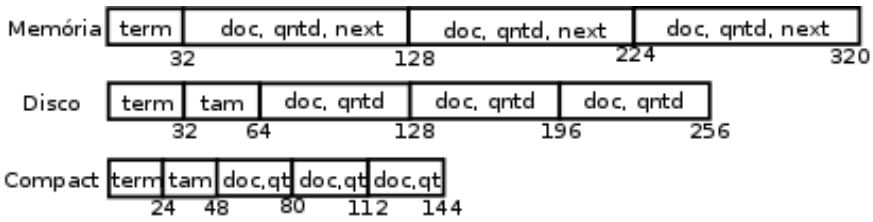

Figura B.1: Exemplo de manipulação manual de bits

A descompressão desses métodos são muito velozes, já que é fácil saber o tamanho real da estrutura simplesmente calculando a partir do campo *tam*. Desse ponto, seria questão apenas de alocar o espaço de memória necessário, uma série de operações de *shift*, extremamente velozes, para ajustar o tamanho correto e a atualização dos ponteiros das listas invertidas.

Os grandes problemas dessa estrutura são a falta de ferramentas prontas (o próprio programador deve criar seus métodos para compressão e descompressão) e a questão de planejamento. No caso apresentado, se o sistema atual indexa 10 milhões de documentos e passar a indexar 50 milhões, será necessário modificar toda a estrutura (e atualizar todos os arquivos), já que os 3 bytes disponibilizados para o identificador do documento referenciam só 16 milhões de documentos.

B.2 Bytewise-compression

Um método largamente utilizado é a compressão baseada em bytes, o *bytewise-compression*. A quantidade de bytes para um certo dado é variável e determinada pela própria estrutura dos bytes do dado. Diversas codificações de caracteres utilizam esse método para reduzir o tamanho de textos, ao mesmo tempo em que suportam uma grande quantidade de caracteres. Nesse sentido, podemos destacar o famoso UTF-8, que possuí 4 diferentes possibilidades de caractere.

Embora até 4 bytes sejam utilizados para um caractere, esses são casos raros. Os caracteres mais comuns são aqueles encontrados no primeiro caso (letras comuns, números,

Bytes	Máscara
1	0*******
2	110***** 10******
3	1110**** 10****** 10******
4	11110*** 10****** 10****** 10******

Tabela B.1: Codificando caracteres UTF-8 com *bytewise-compression*

alguns símbolos), utilizando apenas um byte. Os conjuntos de caracteres especiais (como letras acentuadas ou alfabeto cirílico ou japonês) são representadas com dois ou mais bytes. Na organização do UTF-8, pela máscara do primeiro byte, o sistema já é capaz de identificar com quantos bytes esse caractere é representado, além disso, os bytes iniciados com '10' sempre são um 'byte de continuidade'.

Mas nem só para caracteres é possível se utilizar de compressão baseada em bytes. Para se representar um inteiro de tamanho variável (incluindo os inteiros longos de 8 bytes - ou mais), deve-se olhar o bit de maior ordem do byte. Se ele for '0', esse dado acaba nesse bit, mas se for '1', também deve-se considerar o próximo byte para o valor deste inteiro, e a operação é repetida para este byte. A tabela B.2 exibe os intervalos de valores inteiros utilizando 1, 2 e 3 bytes.

Um problema da utilização de um método deste tipo é a dificuldade em no acesso aleatório a dados. Para recuperar o N-ésimo inteiro de tamanho variável de uma cadeia de bytes, é preciso varrê-la seqüencialmente até o N-ésimo inteiro. Quando todos os dados possuem tamanho fixo, basta se saltar uma quantidade de bytes igual a (N-1) multiplicado pela quantidade de bytes do dado.

B.3 Bitwise-compression

Os métodos de compressão por bits, ou *bitwise-compression*, deixam cada dado com diferentes tamanhos em bits, e apesar de existem uma série de técnicas para isso (como códigos de Huffman, Delta encoding e Gamma encoding), elas são consideradas um tanto lentas, tanto para decodificação do lado quanto devido a não linearidade por bytes, exigindo primeiramente alguns ajustes no dado.

Valor	Byte 1	Byte 2	Byte 3
0	00000000		
1	00000001		
2	00000010		
...			
127	01111111		
128	10000000	00000001	
129	10000001	00000001	
130	10000010	00000001	
...			
16.383	11111111	01111111	
16.384	10000000	10000000	00000001
16.385	10000001	10000000	00000001
...			

Tabela B.2: Codificando um Inteiro com *bytewise-compression*

Apêndice C

Caching

Enquanto a compressão de dados torna as operações de entrada/saída mais velozes, a idéia por trás do *caching* é minimizar a quantidade dessas operações, mantendo na memória dos diferentes níveis de cache aquelas estruturas de dados anteriormente utilizadas para que sejam eventualmente reaproveitadas. É desnecessário dizer que a utilização de compressão de dados em conjunto com *caching* é extremamente eficiente, pois torna a capacidade dos caches maiores.

Bons caches para sistemas de busca geralmente implementam dois sub-caches no mesmo nível: um para as estruturas mais utilizadas naquele nível (determinadas através de uma análise de *logs* do sistema), praticamente estático, pois tem uma freqüência de atualização muito lenta; e outro para as últimas estruturas utilizadas, geralmente utilizando políticas de descarte do tipo LRU (removendo da memória o item do cache que a mais tempo não é utilizado). Dessa forma, são mais rapidamente resolvidas tanto aquelas buscas feitas com maior freqüência (como nomes de celebridades em um buscador web), como aquelas buscas repetidas em curtos espaços de tempo. A figura ao lado mostra a hierarquia dos níveis de cache.

C.1 Result Cache

Este é o nível de cache mais comum e facilmente implementado, além de garantir bons ganhos de desempenho. Ele fica localizado na parte mais externa do sistema, como no servidor web, por exemplo, armazenando o resultado de buscas completas (como "universidade de são paulo"). Quando novas buscas idênticas a essa forem refeitas, o cache evita que essa busca seja refeita para o broker, aliviando a sua carga e aumentando o tempo de resposta do sistema para essas buscas.

Em sistemas baseados em índice global, pode haver uma variação deste nível de cache que fique localizada no broker, armazenando os resultados de *sub-consultas* e não consultas completas. Assim, uma busca por "universidade de são paulo"geraria 3 novas entradas no cache de últimas sub-consultas do broker: para 'universidade', 'são' e 'paulo'. Uma busca por "universidade federal do paraná"não repetiria a sub-consulta por universidade, uma vez que ela já está armazenada neste cache, aliviando a carga para uma das 3 máquinas que seriam envolvidas. Além disso, a combinação de sub-consultas em cache de diferentes consultas pode ser o suficiente para resolver novas consultas.

C.2 List Cache

Este nível de cache foi originalmente desenvolvido para sistemas *disk-based*, onde como o índice é grande demais para ficar inteiramente em memória, uma boa seleção de quais listas manter em memória é crucial para um melhor desempenho desses sistemas. Assim, este nível de caching fica localizado nas unidades indexadoras do sistema.

Com a adição de compressão e caching, é possível que alguns sistemas *disk-based* se tornem praticamente *memory-based*, já que pode ser possível manter todo o índice compactado em memória, e nesse caso seria feito cache para deixar algumas listas não compactadas, aumentando sua velocidade de acesso.

C.3 Projection Cache

O menos utilizado dos níveis de cache, concebido em [25]. Sua função é de armazenar interseções de listas invertidas de duplas de termos que comumente apareçam juntos (como "são paulo", "deputado federal"e "maria josé"). Sua localização seria no broker para índices globais e em cada máquina para índices locais. Entretanto, é fácil imaginar uma implementação em índices globais onde cada servidor teria um *sub projection cache* para as relações entre os termos de responsabilidade deste servidor: um servidor responsável por termos entre as letras O e T, poderia ter a projeção 'são paulo' previamente armazenada neste cache.

Não é tanto utilizado devido a quantidade colossal de termos que aparecem em conjunto, de tal forma que esse cache precisa ser um tanto grande para que tenha utilidade. Devido ao seu tamanho, geralmente é armazenado no disco do broker, já que recuperar uma lista de resultados pronta do disco local é (provavelmente) mais rápido do que repassar as consultas

para os nós do cluster e depois se mesclar os resultados.

Referências Bibliográficas

[1] *Projeto hadoop*, http://hadoop.apache.org, [Online; accessed 10-Fev-2010]. 7, 39

[2] *Projeto lucene*, http://lucene.apache.org/java, [Online; accessed 10-Fev-2010]. 7, 11, 29, 37

[3] *Projeto nutch*, http://lucene.apache.org/nutch, [Online; accessed 10-Fev-2010]. 19

[4] *Projeto solr*, http://lucene.apache.org/solr, [Online; accessed 10-Fev-2010]. 7, 18, 19

[5] *Sciencenet*, http://sciencenet.fzk.de/, [Online; accessed 10-Fev-2010]. 9

[6] *Yacy - distributed p2p-based web indexing*, http://yacy.net/, [Online; accessed 10-Fev-2010]. 8, 19

[7] G. Andrews, *Foundations of multithreaded, parallel, and distributed programming*, first ed., Addison Wesley, 1999. 61

[8] E. Archer and O. Gospodnetic, *Lucene in action*, first ed., Manning Publications, 2004. 7

[9] Claudine Badue, Ricardo Baeza-yates, Berthier Ribeiro-neto, and Nivio Ziviani, *Distributed query processing using partitioned inverted files*, In Proc. of the 9th String Processing and Information Retrieval Symposium (SPIRE, IEEE CS Press, 2001, pp. 10–20. 6, 18, 21, 22, 43

[10] Claudine Badue and Berthier Ribeiro-Neto, *Basic issues on the processing of web queries*, In: Proceedings of the 28th Annual International ACM SIGIR Conference on Reseach and Development in Information Retrieval (SIGIR'05, 2005, pp. 577–578. 6, 19, 21, 43

[11] R. Baeza-Yates, *Searching the world wide web: Challenges and partial solutions*, IBERAMIA '98: Proceedings of the 6th Ibero-American Conference on AI (London, UK), Springer-Verlag, 1998, pp. 39–51. 14

[12] R. Baeza-Yates and B. Ribeiro-Neto, *Modern information retrieval*, first ed., Editora Addison-Wesley, 1999. 15

[13] Sergey Brin and Lawrence Page, *The anatomy of a large-scale hypertextual web search engine*, COMPUTER NETWORKS AND ISDN SYSTEMS, Elsevier Science Publishers B. V., 1998, pp. 107–117. 7, 15, 21, 23, 26, 38, 39

[14] M. Burrows, *The chubby lock service for loosely-coupled distributed systems*, 7th USENIX Symposium on Operating Systems Design and Implementation (OSDI) (2006). 27

[15] Fay Chang, Jeffrey Dean, and Sanjay Ghemawat, *Bigtable: A distributed storage system for structured data*, In Proceedings of the 7th Conference on USENIX Symposium on Operating Systems Design and Implementation, vol. 7, 2006, pp. 205–218. 7, 27, 38

[16] Text REtrieval Conference, D. K. Harman, National Institute of Standards, and Technology (U.S.), *Overview of the third text retrieval conference (trec-3) [microform] / d.k. harman, editor*, U.S. Dept. of Commerce, Technology Administration, National Institute of Standards and Technology, Gaithersburg, MD :, 1995. 6

[17] J. Dean and S. Ghemawat, *Mapreduce: Simplified data processing on large clusters*, Sixth Symposium on Operating System Design and Implementation (OSDI) (2004). 7, 25

[18] Ian Foster, *Globus toolkit version 4: Software for service-oriented systems*, IFIP International Conference on Network and Parallel Computing, Springer-Verlag LNCS 3779, 2005, pp. 2–13. 10

[19] Erich Gamma, Richard Helm, Ralph E. Johnson, and John Vlissides, *Design patterns: Elements of reusable object-oriented software*, Addison-Wesley, Reading, MA, 1995. 42, 45

[20] S. Ghemawat, H. Gobioff, and S. Leung, *The google file system*, 19th ACM Symposium on Operating Systems Principles (2003). 7, 28, 38

[21] Brian Goetz, Tim Peierls, Joshua Bloch, Joseph Bowbeer, David Holmes, and Doug Lea, *Java concurrency in practice*, Addison-Wesley, Upper Saddle River, NJ, 2006. 45

[22] A. Goldchleger, F. Kon, A. Goldman, and M. Finger, *Integrade: object-oriented grid middleware leveraging idle computing power of desktop machines*, Concurrency and Computation: Practice and Experience (2004), 449–454. 10

[23] N. A. B. Gray, *Comparison of web services, java-rmi, and corba service implementation*, Fifth Australasian Workshop on Software and System Architectures, 2004. 38

[24] Matjaz B. Juric, Ivan Rozman, Bostjan Brumen, Matjaz Colnaric, and Marjan Hericko, *Comparison of performance of web services, ws-security, rmi, and rmi-ssl*, The Journal of Systems and Software **79** (2006), no. 5, 689–700. 38

[25] Xiaohui Long, *Three-level caching for efficient query processing in large web search engines*, In Proc. of the 14th Int. World Wide Web Conference, ACM Press, 2005, pp. 257–266. 80

[26] Christopher D. Manning, Prabhakar Raghavan, and Hinrich Schütze, *Introduction to information retrieval*, Cambridge University Press, Cambridge, UK, 2008. 15, 71

[27] Mauricio Marin, Veronica Gil-Costa, Carolina Bonacic, Ricardo Baeza-Yates, and Isaac Scherson, *Sync/async parallel search for the efficient design and construction of web search engines*, (to appear), 2010, To appear. 7, 22, 43

[28] E. Meij and M. Rijke, *Deploying lucene on the grid*, Proceedings of the SIGIR 2006 Workshop on Open Source Information Retrieval (OSIR'2006) (2006). 10

[29] Marco Modesto, Álvaro Pereira, Nivio Ziviani, Carlos Castillo, and Ricardo Baeza-Yates, *Um novo retrato da web brasileira*, Proceedings of XXXII SEMISH (São Leopoldo, Brazil), 2005, pp. 2005–2017. 21

[30] B. Ribeiro-Neto, J. Kitajima, G. Navarro, and N. Ziviani, *Parallel generation of inverted files for distributed text collections*, Proceedings of the XVIII International Conference of the Chilean Computer Science Society (1998). 6, 21

[31] B. Ribeiro-Neto, E. Moura, M. Neubert, and N. Ziviani, *Efficient distributed algorithms to build inverted files*, Proceedings of the 22nd annual international ACM SIGIR conference on Research and development in information retrieval (1999), 105–112. 6, 20, 21