ALGORITMO PARALELO E EFICIENTE PARA O PROBLEMA DE PAREAMENTO DE DADOS

WALTER DOS SANTOS FILHO

ALGORITMO PARALELO E EFICIENTE PARA O PROBLEMA DE PAREAMENTO DE DADOS

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

Orientador: Wagner Meira Junior

Belo Horizonte

Abril de 2008

Santos Filho, Walter dos
S237a Algoritmo Paralelo e Eficiente para o Problema de Pareamento de Dados / Walter dos Santos Filho. — Belo Horizonte, 2008
xxiv, 78 f. : il. ; 29cm

Dissertação (mestrado) — Universidade Federal de Minas Gerais
Orientador: Wagner Meira Junior

1. Pareamento de Registros - Teses. 2. Deduplicação - Teses. 3. Paralelismo - Teses. 4. Algoritmo - Teses. I. Orientador. II Título.

CDU 519.6*73(043)

UNIVERSIDADE FEDERAL DE MINAS GERAIS

PPGCC
PROGRAMA DE PÓS-GRADUAÇÃO EM CIÊNCIA DA COMPUTAÇÃO DCC-UFMG

FOLHA DE APROVAÇÃO

Algoritmo Paralelo e Eficiente para o Problema de Pareamento de Dados

WALTER DOS SANTOS FILHO

Dissertação defendida e aprovada pela banca examinadora constituída pelos Senhores:

PROF. WAGNER MEIRA JÚNIOR - Orientador
Departamento de Ciência da Computação - UFMG

PROFA. CARLA JORGE MACHADO - Co-orientadora
Centro de Desenv. e Planej. Regional - UFMG

PROF. PHILIPPE OLIVIER ALEXANDRE NAVAUX
Instituto de Informática - UFRGS

PROF. DORGIVAL OLAVO GUEDES NETO
Departamento de Ciência da Computação - UFMG

PROF. MARCOS ANDRÉ GONÇALVES
Departamento de Ciência da Computação - UFMG

Belo Horizonte, 22 de abril de 2008.

Aos meus pais, à minha esposa e à minha família, pilares da minha vida

Agradecimentos

Talvez mesmo antes de imaginar o que um dia seria minha dissertação de mestrado, eu já pensava nesta seção. Aqui é possível relembrar pessoas que me ajudaram na vida pessoal, acadêmica e profissional. Gostaria que soubessem que se alcancei este sonho, muito devo a elas.

Primeiramente, agradeço a Deus que sempre me deu forças quando precisei e me deu um lar e possibilidades de ser feliz.

Agradecer minha família me emociona sempre. Obrigado minha mãe, Geralda e meu pai Walter (de quem ainda sinto muita falta). O amor de vocês foi a maior riqueza que tive. Obrigado às minhas irmãs, Eloísa e Sônia, que foram para mim meu horizonte acadêmico e profissional. À Diane, minha esposa, que sempre entendeu minha ausência em alguns momentos, pois ela me compreendia e sabia deste sonho. Obrigado aos meus sobrinhos, Kelly, Jean, Karen e Eric, que me propiciam momentos de diversão, afinal, para ser sério, é preciso se divertir.

Gostaria de agradecer aos mestres que estiveram ao meu lado ao longo de toda minha vida de estudante. Obrigado à Marilene que um dia acreditou que eu poderia ser mais do que eu era. Obrigado ao meu orientador e amigo Wagner Meira, pessoa fantástica em sua inteligência e em querer o bem-estar e desenvolvimento de todos. Se eu tiver que nomear alguém que faz o mundo dar um passo à frente, esse alguém é você, Meira. Obrigado à minha co-orientadora, Carla, por seu apoio e eterna gentileza. Obrigado à Eliane e à Prefeitura Municipal de Belo Horizonte, por terem disponibilizado a base de dados usada nesse trabalho. Obrigado ao Altigran, que mesmo distante, nos ajudou a traçar os rumos de minha dissertação. Obrigado à Mariângela, Augusto e Odilon, do GPES/Faculdade de Medicina da UFMG.

Obrigado aos professores Renato e Dorgival, especialmente pela ajuda nos artigos e discussões sobre meu trabalho. Obrigado também ao professor Antônio Alfredo por sua ajuda ao longo da graduação.

Não há como eu retribuir a ajuda dos amigos Adriano César e Leonardo de Araújo na realização deste momento. Obrigado a vocês pelos momentos de convívio e

por acreditarem em mim. Obrigado aos amigos do *Speed*: Elisa, Fernando Henrique, Leonardo Rocha, Gustavo Orair, Tiago Macambira, Adriano Veloso, Hélio, Arlei, Carlos e especialmente Thiago Teixeira e Charles Gonçalves pela ajuda e companheirismo no desenvolvimento de nossas pesquisas, ao Bruno Coutinho e George pela ajuda com o Anthill e pelo SBAC e finalmente ao André (*Hawks*) por sua ajuda principalmente no Estágio em Docência.

Obrigado novamente ao Leonardo de Araújo, a Rafael Paiva e a Rodrigo Moreira por permitirem que eu me ausentasse da nossa empresa para que eu alcançasse o mestrado.

Obrigado àqueles que acreditaram que este dia chegaria e torceram por mim: pessoal da Grad972, Renato Maia, Eduardo Ostos, Gracielle Ferraz.

A todos vocês, meu mais sincero agradecimento e voto de felicidades.

Resumo

Em um mundo onde cada vez mais a informação se torna importante, contar com bases de dados confiáveis e consistentes é requisito essencial para tomada de decisão, análise de tendências, detecção de fraudes, mineração de dados, suporte a clientes, inteligência de negócio entre outros. Uma das formas de melhorar a qualidade dos dados é eliminar réplicas e consolidar a informação.

Neste trabalho, apresentamos a ferramenta chamada FERAPARDA (FERramenta de Apoio ao PAReamento de DAdos). Ela permite combinar informação de várias bases de dados por meio do pareamento probabilístico de registros. O processo de pareamento se baseia na construção e comparação de pares registros, comparando nomes, endereços e outros atributos que geralmente não serviriam como identificadores individuais e na classificação probabilística do resultado.

Não é raro encontrarmos bases com milhares senão milhões de registros, onde os dados podem apresentar problemas como ausência, inconsistência, erros de entrada ou mesmo duplicidade de informação. Tais problemas e a quantidade de registros obrigam a comparação de muitos pares (no pior caso, quadrático em relação ao tamanho da base), algo que torna o processo muito demorado para ser executado em um único computador. Geralmente, o processo de pareamento de registros é executado mais de uma vez com seus parâmetros sendo ajustados a cada execução, uma vez que características da base de dados podem tornar difícil a decisão sobre o resultado. Um exemplo são bases de dados onde nomes de pessoas ocorrem com grande freqüência ou ainda situações onde é muito difícil diferenciar se dois registros dizem respeito à mesma pessoa, como é o caso de gêmeos.

Existem muitas ferramentas que realizam o pareamento probabilístico de registros. No entanto, poucos trabalhos discutem a paralelização do processo, que se torna ainda mais necessária quando lidamos com bases de dados reais. Para diminuir o tempo de processamento, estudamos neste trabalho formas de paralelizar o algoritmo de pareamento de registro. Apresentamos e discutimos cada etapa do processo de pareamento e como ele foi paralelizado. Conseguimos com sucesso implementar uma

solução capaz de escalar bem quando executada em um *cluster* de computadores.

Neste trabalho também discutimos diferentes aspectos do paralelismo aplicados ao problema e também como a localidade de referência pode ser explorada a fim de maximizar o desempenho e escala da implementação, sem no entanto demandar uma grande quantidade de recursos, especialmente memória principal. Mostramos como o uso de cache de comunicação é fundamental para a escalabilidade e como uma das etapas - a blocagem - tem importância direta neste resultado.

Esperamos que a ferramenta FERAPARDA possa ser usada em diferentes bases de dados, desde bases comerciais até bases da saúde e de programas sociais a fim de melhorar a qualidade da informação e melhorar a qualidade dos serviços que se baseiam em tal informação.

Abstract

In a world where the information is becoming more important each day, the availability of reliable and consistent databases is essential for decision-making, trend analysis, fraud detection, data mining, customer support, and business intelligence, among other data-intensive applications. In order to sustain data quality standards, it is frequently necessary to discard replicas and consolidate the information.

In this work we introduce a tool named FERAPARDA (from the Portuguese acronym for "tool for record linkage"). It allows the combination of information from several sources through probabilistic record linkage. The linkage process is based on building and comparing pairs of records in a per attribute basis, that is, matching names, addresses and other attributes that are not unique identifiers, and finding replicas probabilistically.

Large databases containing thousands and even millions of records are quite common, and they usually present several problems such as missing and inconsistent data, input errors or even replicated information. These problems and the database size result in a need for comparing a large number of pairs of records (presenting a quadratic complexity in the worst case), making the process laborious and time-consuming for the execution in a single machine. Generally, the linkage process is calibrated iteratively, as a consequence of database characteristics, such as very frequent names or challenging pseudo-replicas, such as records from twins.

There are several tools that perform probabilistic linkage of records. However, few efforts discuss the process parallelization, what is even more importante for real datasets. In order to reduce the execution time, we discuss parallelization strategies of the record linkage algorithm. We present and discuss each step in the linkage process and how it was parallelized. We were succesful in the sense that our solution scales well in computing clusters.

This work also discusses various parallelization issues applied to the problem and how the reference locality may be exploited towards maximizing performance without requiring a large amount of resources, in particular memory. We show that the usage

of a communication cache is key for the scalability of the algorithm and how one of the linkage steps, blocking, is fundamental in this work.

We believe that FERAPARDA is capable of performing the linkage of various databases, from commercial to health records, enhancing the quality of the data and the services that are based on that information.

Lista de Figuras

Lista de Tabelas

Lista de Algoritmos

Sumário

Capítulo 1

Introdução

O volume de dados gerados e armazenados por organizações e empresas tem crescido cada vez mais nos últimos anos, bem como a preocupação com a qualidade desses dados. Informação imprecisa pode levar a decisões errôneas e, por isto, manter bases consistentes e confiáveis é fundamental. Mas nem sempre isto é possível. Bases reais geralmente são alimentadas com registros provenientes de procedimentos administrativos, questionários, reconhecimento através de OCR, extração de dados de mídias eletrônicas e inserção manual. Durante a transcrição, digitação ou mesmo armazenamento dos dados, é muito provável que sejam introduzidos erros e variações em algum momento. Por exemplo, em bibliotecas digitais como a ACM, DBLP, Google Acadêmico e Citeseer, é comum encontrarmos variações na escrita do nome de autores, conferências e mesmo título dos artigos [Kan & Tan, 2008] em decorrência do processo de OCR e mesmo porque não é seguida uma única forma de escrita de nomes, títulos e conferências.

Consideramos uma *entidade* como sendo uma representação de um conceito real em determinado contexto. Como exemplos de entidades, podemos citar autores, artigos e conferências em bibliotecas digitais, pacientes, médicos e hospitais em bases da saúde ou ainda clientes e produtos em bases comerciais. Uma mesma entidade pode ter parte de seus dados segmentados em duas ou mais bases. Por exemplo, uma mesma pessoa pode ter seus dados pessoais registrados em um cadastro escolar, em prontuários hospitalares, em cadastros bancários, entre outros. Damos o nome de *pareamento de registro* (*record linkage*) ao processo de combinar informações de uma mesma entidade que se encontram segmentadas em várias bases de dados [Winkler, 2006].

Uma variação do problema de pareamento de registro ocorre quando temos uma única base e há duplicidade de informação como, por exemplo, um mesmo cliente cadastrado duas vezes. Das diversas variações de uma mesma entidade, podemos escolher uma (arbitrariamente ou não) como sendo a *canônica*, enquanto todas as outras

seriam as suas *réplicas*. Em muitos casos, essas réplicas podem degradar o desempenho e a confiança de uma base de dados e por isto devem ser eliminadas. Ao processo de eliminar réplicas em uma base de dados dá-se o nome de *deduplicação*.

O pareamento de registros envolve comparar pares de registros e avaliar se ambos se referem à mesma entidade. No caso extremo, todos os registros serão comparados contra todos os outros, resultado em $(n-1) \times (n-2)/2$ comparações no processo de deduplicação e $m \times n$ comparações no processo de pareamento de registros. Existem técnicas (apresentadas na seção 3.2) que se propõem a diminuir o número de comparações mas, ainda assim, esse número pode ser bem grande, o que torna o processamento paralelo uma escolha quando precisamos diminuir o tempo de processamento.

1.1 Motivação

O problema de *pareamento de registros* é encontrado nas mais diferentes áreas. Na área de saúde, diversos trabalhos [Drumond & Machado, 2008; Cherchiglia et al., 2007] têm usado técnicas de pareamento de registros com a finalidade de integrar bases de dados diferentes e extrair informações para estudo de políticas de saúde. Trabalhos sobre qualidade das bibliotecas digitais [Kan & Tan, 2008] discutem a consolidação de nomes de autores e eliminação de informação incorreta ou redundante. O Censo americano vem utilizando o pareamento de registro há anos para melhorar a qualidade dos dados [Jaro, 1989] e inclusive reduzir custos com o próprio levantamento dos dados, dispensando visitas de entrevistadores quando a informação pudesse ser obtida de outra fonte [Winkler, 2006]. Empresas privadas têm economizado milhões de dólares ao resolver problemas de estoque e logística ao deduplicar suas bases de dados [Kan & Tan, 2008].

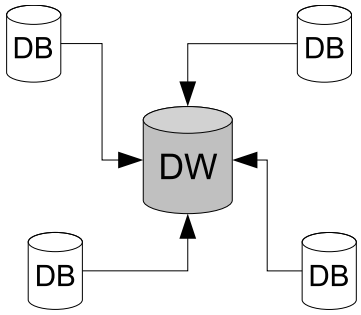


Figura 1.1. Cenário de Armazéns de Dados onde diferentes bases devem ser consolidadas

O volume dos dados é a principal motivação técnica para a criação de um algoritmo para a paralelização do problema de pareamento de registros. Dados fornecidos

pelo Datasus [Datasus, 2008] mostram que de 1995 a 2007, foram realizadas quase 160 milhões de internações hospitalares no Brasil. Em 2004, um levantamento realizado pelo Governo Federal descobriu que existiam 541 milhões de registros de cidadãos inscritos nos cadastros sociais e que, destes, 289 milhões foram facilmente identificados como réplicas. Outros 252 milhões de registros excedem em muito o número de habitantes do país e portanto, existem mais réplicas [Serpro, 2004]. Pelo que conhecemos, existem na literatura poucos estudos sobre a paralelização da deduplicação de registros [Peter Christen, 2004; Kawai et al., 2006; Lee & Kim, 2007]. Nestes estudos, os detalhes sobre a paralelização não estão claros e mesmo as bases de dados utilizadas para os testes são muito pequenas se comparadas às bases reais.

1.2 Objetivos

Este trabalho tem como objetivo entender o processo de pareamento de registros e implementar um arcabouço extensível que suporte a execução das várias etapas do processo em paralelo, sendo uma continuação do trabalho apresentado pelo autor Santos et al. [2007].

1.3 Contribuições

1. Este trabalho apresenta a proposta de um algoritmo paralelo e eficiente e sua implementação para o problema de pareamento de registros. Apresentamos como cada etapa do processo foi paralelizada e discutimos as principais decisões de implementação.

2. Foi construída uma ferramenta, chamada FERAPARDA, que permite realizar o pareamento de registros utilizando diferentes funções de comparação e de codificação de caracteres. A ferramenta também poderá ser estendida para contemplar outras técnicas de blocagem, de pareamento e também novas etapas no processo.

3. Discutimos como explorar a localidade de referência no problema. Implementamos uma *cache* de comunicação que reduziu significativamente a necessidade de troca de registros. Avaliamos também como a blocagem influencia na localidade de referência e que, ao final, uma *cache* pequena é suficiente para processar milhões de pares, mantendo uma boa escalabilidade.

4. A ferramenta FERAPARDA, resultado deste trabalho, foi aplicada a uma base real em um estudo conduzido junto à Secretaria Municipal de Saúde de Belo

Horizonte. O trabalho ajudou na identificação de subnotificações de óbito em Belo Horizonte.

5. Apresentamos também o estado da arte dos trabalhos relacionados a pareamento de registros e como esse problema tem sido abordado no tocante ao processamento paralelo.

1.4 Organização

Essa dissertação está dividida da seguinte forma: o capítulo 2 apresenta os conceitos e etapas do pareamento de registros; o capítulo 3 apresenta uma visão geral dos trabalhos relacionados; o capítulo 4 apresenta o algoritmo de pareamento de registros em paralelo, discutindo suas decisões, oportunidades de paralelização exploradas e como foi feita sua implementação. O capítulo 5 avalia a primeira implementação do algoritmo; o capítulo 6 apresenta a extensão da implementação original para suporte a *caches* e discute como explorar a localidade de referência. O capítulo 7 mostra os resultados obtidos com a utilização de *caches* e, finalmente, o capítulo 8 apresenta as conclusões e trabalhos futuros.

Capítulo 2

Pareamento de Registros

Neste capítulo, são apresentados os principais conceitos aplicados ao trabalho, especialmente aqueles relacionados ao pareamento de registros.

A seção 2.1 apresenta a definição de pareamento de registro. A seção 2.2 apresenta as etapas do processo.

2.1 O Problema do Pareamento de Registros

O problema de pareamento de registro é conhecido por vários nomes na literatura: *record linkage*, *deduplicação*, *entity resolution*, *merge-purge problem*, entre outros [Winkler, 2006]. Neste trabalho, decidimos utilizar o termo *pareamento de registro* para o processo de combinar informações de várias bases de dados que se referem a uma mesma entidade. Neste caso, podem existir relações 1-para-1, como no caso de associação de um registro de um nascido vivo e seu registro de óbito ou ainda 1-para-muitos, como registro de paciente e suas internações. O termo adotado é *deduplicação* quando se faz referência ao processo particular de pareamento de registros que visa eliminar réplicas. Esta deduplicação pode ser interna (uma única base de dados) ou não (eliminação de réplicas por meio da junção de duas ou mais bases de dados sendo mescladas).

Se existe algum identificador ou chave formada por um conjunto de atributos disponível em todas as bases de dados sendo pareadas, o processo de pareamento de registros é trivial, bastando uma operação de *join* em SQL ou algo equivalente. Entretanto, em muitos casos, esse identificador não existe ou não é disponibilizado por questões de confidencialidade e técnicas mais sofisticadas necessitam ser usadas. Essas técnicas podem ser classificadas em dois grandes grupos: *determinista* (ou baseada em regras) e *probabilística* (baseada em probabilidades de concordância e de discordância em pares sabidamente corretos e incorretos). O pareamento de registro determinista

pode ser aplicado caso exista um conjunto de atributos que formam uma *chave de ligação*. Para obter bons resultados, essa chave de ligação deve ser formada por atributos precisos, robustos, estáveis ao longo do tempo e presentes em todas as bases de dados envolvidas. Uma alternativa às chaves de ligação é o uso de um conjunto de regras. O uso desse conjunto de regras flexibiliza o pareamento, mas pode ser de difícil elaboração. Na prática, o pareamento de registros determinista é viável apenas em pequenas bases de dados e resultados empíricos mostram que seus resultados são piores do que o pareamento probabilístico [Christen & Goiser, 2007].

A técnica probabilística tradicional [Fellegi & Sunter, 1969] utiliza os atributos comuns das entidades, tais como nomes, endereços e datas para identificar pares de registros reais. Esses atributos podem conter erros na escrita, estarem em formatos inconsistentes, abreviados, desatualizados ou mesmo ausentes. Um par é considerado verdadeiro (*match*) se os atributos comuns predominantemente casam entre si e é considerado falso (*non-match*) se os atributos comuns predominantemente discordam.

2.2 O Processo de Pareamento de Registros

O processo de pareamento de registros pode ser dividido em etapas, como pode ser visto na Figura 2.1.

2.2.1 Limpeza e Padronização e Análise dos Dados

As primeiras etapas do processo de pareamento de registros são a limpeza e a padronização. Essas duas etapas convertem os dados de entrada de formato bruto para dados bem-formados e consistentes na medida do possível. É comum encontrarem-se casos onde a informação é bastante precária ou não é confiável e o melhor a se fazer é considerá-la como ausente.

A análise dos dados irá identificar os parâmetros a serem usados nas etapas de blocagem, comparação e classificação. Esta análise, regra geral, é feita por um especialista, ou os parâmetros são gerados por um algoritmo de aprendizado [de Carvalho et al., 2006].

Apesar de fazer parte do processo de pareamento de registros, essas etapas não serão abordadas neste trabalho, uma vez que foram consideradas relativamente simples do ponto de vista de paralelização e, geralmente, são realizadas apenas uma vez.

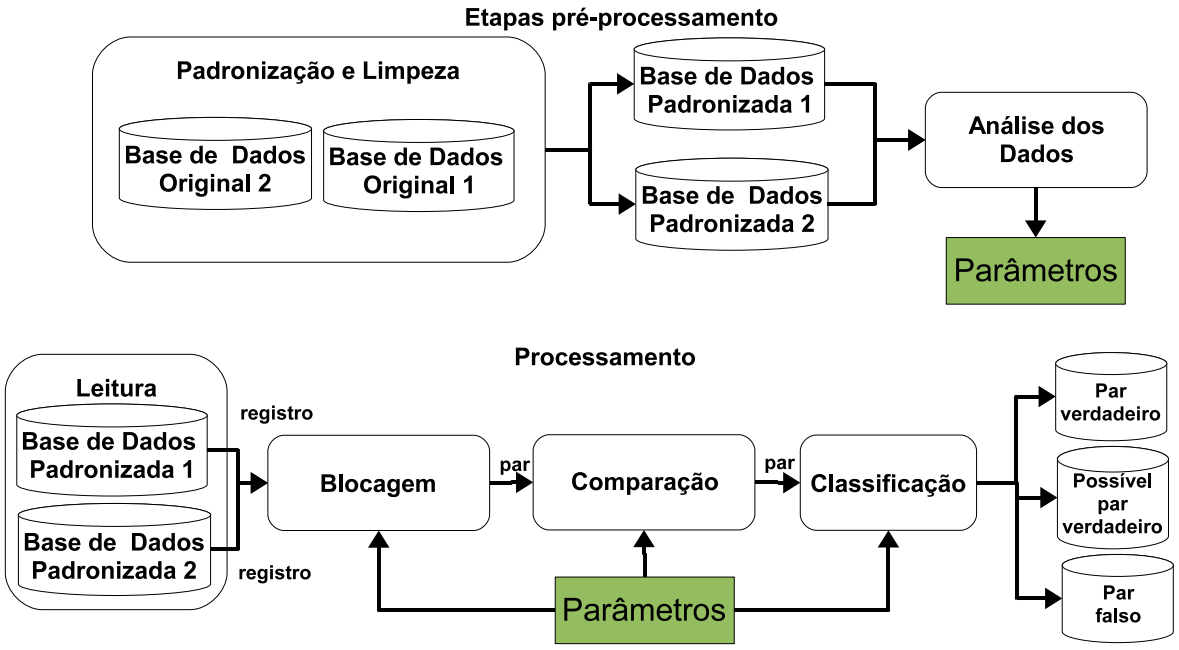


Figura 2.1. O processo de pareamento de registros

2.2.2 Blocagem

A blocagem (do inglês - *blocking*), também conhecida como indexação [Peter Christen, 2004], tem como objetivo limitar o número de comparações. Considerando-se a deduplicação, o número máximo de comparações é $|A| \times (|A| - 1)/2$ e para o pareamento de registros é $|A| \times |B|$, sendo $|A|$ e $|B|$ a quantidade de registros nas bases de dados. Entretanto, pode-se perceber que a maior parte das comparações são supérfluas. Como exemplo, para duas bases de dados com 10 mil registros cada, teremos 100 milhões de comparações no pareamento de registros no caso extremo. Assumindo que a relação entre duas bases de dados é unívoca, o número máximo de pares verdadeiros é dominado pelo tamanho da menor base. Portanto, o número de pares candidatos cresce de forma quadrática, mas o número de pares reais cresce linearmente [Baxter et al., 2003].

Diversos trabalhos (ver Capítulo 3) discutem diferentes estratégicas de blocagem. Neste trabalho, foi implementada a blocagem clássica. Para definirmos quais os atributos dos registros e quais transformações serão aplicadas na blocagem, utiliza-se o conceito de predicado de blocagem [Hernandez & Stolfo, 1998]. Um predicado é *uma disjunção de conjunções*, onde cada termo da conjunção define uma função de transformação sobre o registro. De forma simplista, um predicado de blocagem tem sua definição semelhante à definição da cláusula *where* de SQL. Um exemplo de predicado é $P = (nome \wedge ano\_de\_nascimento) \cup (sobrenome \wedge cidade)$.

Quando aplicado a um registro, o predicado de blocagem é capaz de gerar uma

chave de blocagem para cada conjunção. Registros que geraram a mesma chave de blocagem farão parte de um *bloco* e as comparações são realizadas apenas entre registros do mesmo bloco (ver Figura 2.2). No exemplo, o predicado de blocagem é definido como a concatenação do ano e da cidade, gerando a chave *1*977NY. Os identificadores dos registros que já foram processados estão listados dentro do bloco (*1*, 43, 53, ... e 87). Quando um novo registro com identificador *3*23 é lido a partir da entrada, uma chave de blocagem com valor *1*977NY é gerada e o bloco com a mesma chave é recuperado. Forma-se então um produto cartesiano entre o conjunto de registros já existentes no bloco e o registro recém-lido.

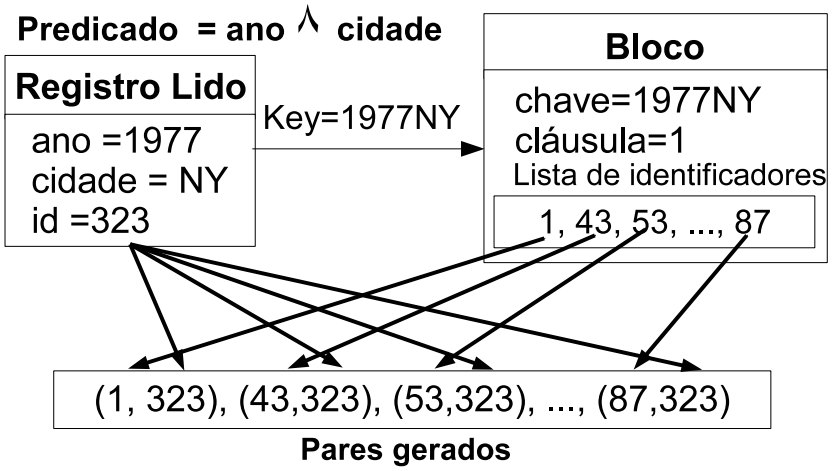


Figura 2.2. Geração de pares na blocagem

A definição do predicado de blocagem deve considerar dois aspectos principais que influenciam diretamente o número de pares gerados e a qualidade do resultado final:

1. Erros nos atributos: Erros nos atributos podem prejudicar a qualidade dos pares gerados, ao levar à exclusão de pares que seriam, de fato, verdadeiros. Por isto, a escolha dos atributos deve levar em conta a qualidade da informação ali contida.

2. Freqüência dos valores dos atributos: Alguns atributos podem ter poucos valores e por isto, serem pouco discriminativos. Por exemplo, o atributo sexo é pouco discriminativo, pois, geralmente, ele segmenta a base de dados em apenas dois valores. Nomes comuns, como *Maria* e sobrenomes como *Silva* podem, também, dominar a geração dos pares.

Por fim, existe um compromisso quando se define o predicado de blocagem. Um predicado mais restritivo conduz à geração de um número maior de pequenos blocos.

Neste caso, os erros, ainda que pequenos, poderão fazer com que pares verdadeiros sejam excluídos da comparação. Por outro lado, com um predicado menos restritivo teremos poucos blocos de tamanho maior, possivelmente cobrindo mais pares verdadeiros, mas com crescimento importante do número de pares totais gerados e aumento no tempo de processamento.

2.2.3 Comparação

A etapa de comparação utiliza os pares gerados pela etapa de blocagem e, em seguida, produz um resultado numérico associado à comparação dos atributos dos registros. A comparação pode ser determinista ou probabilística, conforme observado na seção 2.1. Neste trabalho, apenas a probabilística é considerada, muito embora a determinista possa ser facilmente implementada.

A função de comparação de atributos é baseada em probabilidades de concordância e discordância em pares verdadeiramente corretos ou incorretos. A fim de definir se um par gerado é correto ou incorreto, podem ser utilizadas algumas funções auxiliares de comparação de atributos, que podem ser bastante simples, como comparação exata de caracteres ou números, ou podemos utilizar funções que levam em conta erros tipográficos ou ainda variações fonéticas.

Neste caso, cada função de comparação retornaria um valor numérico, o qual pode ser normalizado entre *zero* e *um*, onde zero indica que os valores dos atributos são totalmente diferentes e um valor acima de um mínimo pré-estabelecido indica concordância. O valor mínimo para concordância é um fator de ajuste. Quanto mais próximo de zero, maior a tolerância a erros. A tabela 2.1 mostra alguns valores para a comparação de nomes usando diferentes funções.

Nome 1	**N**ome 2	**J**aro	**W**inkler	**D**ist. Edição	**B**igram
jose	josue	0.9330	0.9530	0.8000	0.5710
mariana	adriana	0.9050	0.9050	0.7140	0.6670
homero	rogerio	0.6210	0.6210	0.5710	0.3640
karla	carla	0.8670	0.8670	0.8000	0.7500
geralda	geralda	1.0000	1.0000	1.0000	1.0000
cibele	sibele	0.8890	0.8890	0.8330	0.8000
felipe	phillippe	0.6200	0.6200	0.4440	0.4620
vander	wander	0.8890	0.8890	0.8330	0.8000
maria	marta	0.8670	0.9070	0.8000	0.5000

Tabela 2.1. Exemplos de funções de comparação de strings

2.2.4 Classificação

A última etapa, *classificação*, considera alguma função de similaridade que sumariza os resultados da comparação do par de registro e classifica-o em um *par verdadeiro*, um *par falso*, ou um *par possível* (não classificado como verdadeiro ou falso).

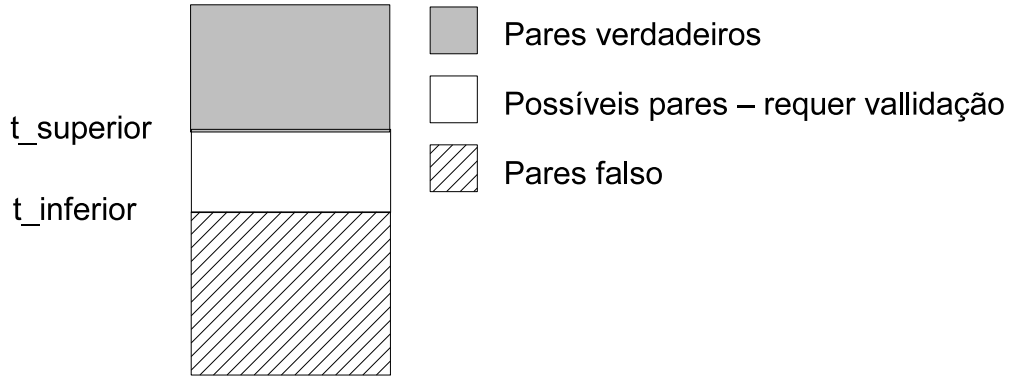


Figura 2.3. Faixas de classificação dos pares comparados

Neste trabalho, aplica-se uma discriminação dos pares conforme sugerida por Fellegi e Sunter. Declara-se (ou considera-se) um par como verdadeiro quando a soma dos resultados das comparações dos atributos é superior a um limiar $T_{superior}$. Do mesmo modo, um par é declarado (ou considerado) falso quando a soma dos resultados das comparações dos atributos é inferior a um limiar $T_{inferior}$. Um par é declarado (ou considerado) com um par possível quando a soma dos resultados das comparações dos atributos situa-se entre os limiares $T_{inferior}$ e $T_{superior}$, como pode ser visto na Figura 2.3. Os valores de $T_{inferior}$ e $T_{superior}$ são parâmetros para o processo e podem ser obtidos pela análise dos dados ou por algum algoritmo de aprendizado.

A região de classificação de pares como possíveis idealmente deverá ser a menor possível. Nessa região encontram-se pares verdadeiros e pares falsos que não puderam ser discriminados pelas funções de comparação. Dois trabalhos, neste caso, poderão ser feitos: análise manual dos pares ou, então, refinar o pareamento para essa região, mudando-se os parâmetros de comparação.

A Figura 2.4 mostra um exemplo do resultado final obtido na comparação de registros de uma base sintética. O histograma dos pesos representa a soma de resultados numéricos de concordâncias e discordâncias, resultantes da comparação dos pares. As duas barras verticais em $x = 19$ e $x = 31,5$ representam respectivamente $T_{inferior}$ e $T_{superior}$.

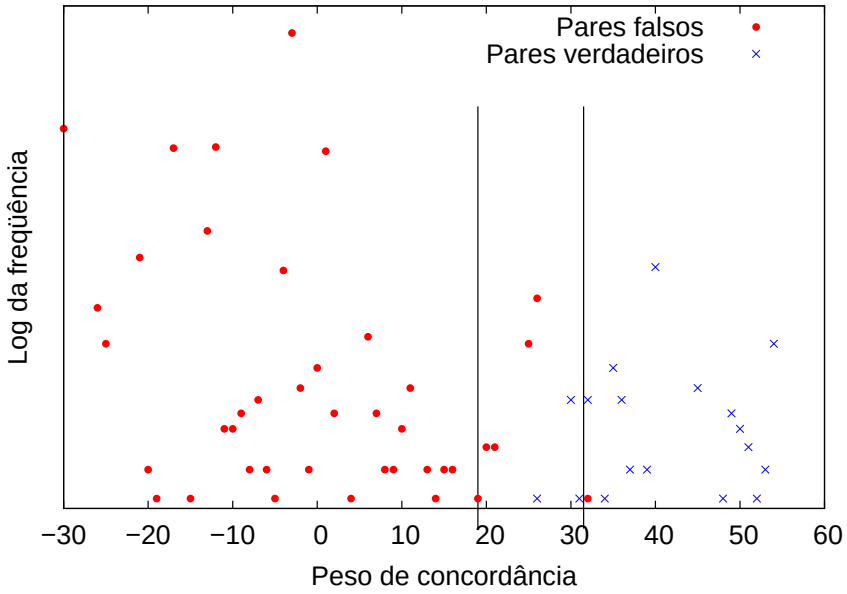


Figura 2.4. Faixas de classificação dos pares comparados

2.3 Algoritmos

O algoritmo seqüencial para o processo de pareamento de registros é a sua representação direta, como pode ser visto nos Algoritmos 2 (duas bases) e 1 (uma única base - interno). A diferença entre eles está na geração dos pares candidatos.

```
1  PareamentoUmaBase (Base, Config, Predicado):
2  HashTable ← ∅
3  foreach Registro ∈ Base do
4      foreach Conjunção ∈ Predicado do
5          Key ← ∅
6          foreach funcaoTransformar ∈ Conjunção do
7              Key ← concatenar(Key, funcaoTransformar(Registro))
8          /*Cria um novo bloco*/
9          if Block ← get(HashTable, Key) = NULL then
10             Block ← createBlock()
11             put(HashTable, Key, Block)
12         /*Comparação*/
13         foreach OldRec ∈ Block do
14             Pair ← (Record, OldRec)
15             Result ← compare(Configuration, Pair)
16             isPair ← classify(Result)
17         Block.Append(Record)
```

Algoritmo 1: Algoritmo seqüencial para uma base de dados

```
1  ParеamentoDuasBases (BaseMenor, BaseMaior, Config, Predicado):
2  HashTable ← ∅
3  foreach Registro ∈ BaseMenor do
4      foreach Conjunção ∈ Predicado do
5          Key ← ∅
6          foreach funcaoTransformar ∈ Conjunção do
7              Key ← concatenar(Key, funcaoTransformar(Registro))
8              /*Cria um novo bloco*/
9              if Block ← get(HashTable, Key) = NULL then
10                 Block ← createBlock()
11                 put(HashTable, Key, Block)
12             Block.Append(Record)
13 foreach Registro ∈ BaseMenor do
14     foreach Conjunção ∈ Predicado do
15         Key ← ∅
16         foreach funcaoTransformar ∈ Conjunção do
17             Key ← concatenar(Key, funcaoTransformar(Registro))
18             /*Recupera um bloco existente*/
19             if Block ← get(HashTable, Key) ≠ NULL then
20                 /*Comparação*/
21                 foreach OldRec ∈ Block do
22                     Pair ← (Record, OldRec)
23                     Result ← compare(Configuration, Pair)
24                     isPair ← classify(Result)
```

Algoritmo 2: Algoritmo seqüencial para duas bases de dados

Quando duas bases são utilizadas, uma otimização de memória implica gerar todos os blocos a partir da menor base de dados em número de registros(linhas 3-12 , algoritmo 2). Em seguida, cada registro da base maior é lido e assim que são formados os pares, esse registro pode ser descartado. Caso a chave de blocagem gerada para o registro da maior base não tenha correspondente na base menor, nenhum par será gerado (linhas 13-24) , algoritmo 2). Esta função lê os dados do registro e transforma um ou mais atributos em parte da chave de blocagem a ser utilizada. Exemplos de funções de transformação incluem as funções de codificação de caracteres, usadas para diminuir os problemas causados por erros nos registros.

Capítulo 3

Revisão Bibliográfica

Feita a introdução sobre os conceitos relacionados ao pareamento de registros, apresenta-se neste capítulo uma visão geral das pesquisas relacionadas ao presente trabalho. A seção 3.1 cita os principais trabalhos que formalizaram a abordagem probabilística para o problema de pareamento de registros. A seção 3.2 apresenta as principais técnicas de blocagem utilizadas hoje e os trabalhos que discutem a aplicação, benefícios bem como os problemas e limitações de cada técnica. O estudo dessas técnicas foi importante para o trabalho, uma vez que permitiu avaliar os principais requisitos para que novas técnicas fossem incluídas na aplicação. Além disto, como será discutido no capítulo 6, a blocagem terá papel fundamental quando localidade de referência é explorada. Finalmente, a seção 3.3 apresenta as algumas ferramentas usadas em pesquisas, limitando-se apenas àquelas que suportam processamento paralelo.

3.1 Pareamento de Registros

O pareamento de registros vem sendo utilizado há muitos anos, com o objetivo de eliminar réplicas ou unificar, por meio de associações, registros que estejam contidos em bases de dados distintas. Entre as bases de dados que podem ser pareadas estão as de dados médicos sobre pacientes; dados do Censo; dados sobre contribuintes ou benefícios do Governo, entre outros.

Em seu trabalho, Gill [Gill, 2001] afirma que, apesar de existirem alguns estudos sobre o pareamento de registros feitos na segunda metade do século XIX e primeira metade do século XX, foi a partir de 1950 que estudos mais confiáveis surgiram. As primeiras análises a tratar o pareamento probabilístico de registros [Newcombe, 1967][Acheson, 1968] avaliaram a viabilidade da aplicação da técnica. Nestes, utilizaram-se pesos obtidos essencialmente de forma empírica, por meio de inspeção

manual de freqüência de acertos em pares sabidamente verdadeiros. A elaboração de uma sustentação teórica para o pareamento probabilístico foi feita nos fins da década de 1960, com os trabalhos de Nathan (1967) [Nathan, 1967], Tepping [Tepping, 1968], D'Andrea Du Bois [N. S. D'Andrea Du Bois, 1969] e de Fellegi e Sunter [Fellegi & Sunter, 1969]. Este último se tornou o principal trabalho da área, tornando-se também a referência definitiva sobre pareamento probabilístico de registros.

3.2 Blocagem

Os métodos de blocagem buscam selecionar de forma eficiente um subconjunto dos pares de registros a serem comparados em etapas posteriores do processo. Diferentes técnicas têm procurado diminuir a quantidade de pares candidatos e cobrir o maior número possível de pares reais.

A *b*locagem clássica ou *p*adrão agrupa todos os registros que geraram a mesma chave em um bloco; apenas registros dentro do mesmo bloco são comparados entre si. Em sua definição, um registro será inserido apenas dentro de um único bloco. Sua implementação é simples e eficiente quando se utiliza um índice invertido [Baxter et al., 2003]. Cada chave de blocagem cria uma entrada nesse índice assim que é gerada pelo primeiro registro. Cada novo registro que gera uma chave existente é inserido no índice. Ao final da leitura dos registros, a lista de registros é extraída da lista invertida e formam-se os pares a partir de cada bloco.

Uma das principais limitações associadas a esta técnica de blocagem se refere à inserção de um registro em um bloco incorreto, comum quando existem erros de entrada. Para evitar esse tipo de problema, podemos utilizar funções de codificação de caracteres ou, ainda, usar mais de uma opção de blocagem.

Hernandez et al. [Hernandez & Stolfo, 1998] propuseram uma blocagem baseada na utilização uma lista invertida, onde as entradas são ordenada pela chave de blocagem. A premissa básica é a de que uma janela deslizante de tamanho fixo $w > 1$ move-se seqüencialmente sobre os registros ordenados e todos aqueles que estiverem dentro da mesma janela serão comparados. A principal vantagem desta técnica é que a quantidade final de pares pode ser controlada. Ao final, cada registro gerará *2w - 1* pares para comparação, resultando um total de $O(nw)$ pares em uma base de dados com n registros no total [Baxter et al., 2003]. A desvantagem é que para o seu funcionamento correto, essa técnica assume que a ordenação não apresenta erros (especialmente erros no primeiro caractere da chave). Contudo, podemos superar esta limitação: caso haja suspeita de erros, podemos definir uma combinação de chaves e utilizar funções

de codificação de caracteres.

Uma outra técnica de blocagem é a *Q*-gramas [Baxter et al., 2003]. A blocagem *Q*-gramas permite que pequenas variações nos valores das chaves (inclusão, alteração e remoção de caracteres) não influenciem o resultado final. Nesta técnica, cada registro é inserido em mais de um bloco. A chave é transformada em uma lista de *q-gramas* (divisões da chave em partes de q caracteres) e todas as combinações desses *q-gramas* acima de um certo limiar t são criadas. Em seguida, os *q-gramas* que compõem cada combinação são novamente concatenados e usados como chave da lista invertida. Por exemplo, para uma chave original de blocagem *'silva'*, com $q = 2$ (isto é, bigrama) e limiar $t = 0,8$, temos a seguinte lista de bigramas: $['si','il','lv','va']$. O tamanho da lista e o limiar são utilizados com a finalidade de determinar as combinações a serem geradas. No caso, o tamanho da lista é multiplicado pelo limiar, resultando em $4 \times 0,8 = 3,2$, arredondando, 3. Isto implica que todas as combinações de tamanho maior ou igual a 3 serão consideradas: $['si','il','lv','va'], ['si','il','lv'], ['si','il','va'], ['si','lv','va']$ e $['il','lv','va']$. As chaves inseridas na lista invertida serão, assim, *'siillvva', 'siillv', 'siilva', 'silvva' e 'illvva'*. O tamanho da chave de blocagem impacta diretamente esta técnica, podendo levar a uma explosão no número de combinações.

Bilenko, Kamath e Mooney discutem em seu trabalho [Bilenko et al., 2006] técnicas adaptativas para a blocagem, visando melhorar a escalabilidade do pareamento. Segundo os autores, as técnicas mais utilizadas em anos recentes, baseadas na utilização de funções de similaridade ou na indexação, requerem um ajuste manual e fino, para que se minimize número de pares falsos e maximize o número de pares positivos. Deste modo, os autores apresentam uma abordagem capaz de gerar, a partir de uma base de treinamento, predicados de blocagem na forma normal disjuntiva (*disjunctive normal form - DNF*).

Ainda sobre a blocagem, Bilenko discute em sua dissertação [Bilenko, 2006] o uso de algoritmos de *clustering*, especificamente o *K-Means*, obtendo resultados interessantes sobre funções de similaridade aplicadas tanto na blocagem, quanto na comparação de registros.

Baxter e Gu [Gu & Baxter, 2004] apresentam em seu trabalho o conceito de filtros adaptativos para a blocagem. Para estes autores, por melhor que seja a chave de blocagem, quase sempre haverá blocos muito grandes. Por exemplo, para o idioma inglês, *Smith* e *Taylor* são sobrenomes comuns. A idéia seria então reprocessar todos os blocos que são considerados grandes, realizando uma espécie de filtragem dentro da blocagem, para eliminar pares não relevantes, ou seja, claramente não referentes ao mesmo indivíduo ou entidade. Assim, informações semânticas podem auxiliar essa técnica e são de extrema importância. Por exemplo certas doenças como o câncer de

útero não são aplicáveis a um dos sexos.

3.3 Ferramentas

O Febrl [Peter Christen, 2004] é uma das mais completas ferramentas de pareamento de registros disponibilizadas como software livre. Esta ferramenta implementa a abordagem clássica para o pareamento de registros [Fellegi & Sunter, 1969] e foi construído como projeto de pesquisa da *Australian National University* para pareamento de registros da área médica. Ao longo dos anos, a ferramenta vem sendo estendida e aprimorada com várias técnicas de blocagem (clássica, *sorting blocking*, *q-grams*), funções de comparação (incluindo informações geo-espaciais), tabelas de correção de erros (*lookups*) e processamento paralelo. Um dos autores do Febrl, Christen, discute em seu trabalho [Christen, 2005] a geração de bases sintéticas para o pareamento de registros e inclui no Febrl a implementação da ferramenta DsGen. Esta ferramenta gerou as bases sintéticas usadas nos primeiros experimentos desta dissertação.

Especificamente sobre o paralelismo de dados o Febrl utiliza uma interface de programação escrita na linguagem Python e que abstrai as chamadas às funções da biblioteca MPI [Gropp et al., 1996]. Em vários testes realizados com bases reais, o Febrl não suportou mais do que 3 milhões de pares. Sempre ocorria um problema de falta de memória, ainda que o computador dispusesse de 2GB de memória principal. Além disto, por ser totalmente escrito em Python - uma linguagem interpretada - o desempenho geral do Febrl não é bom, sendo três ordens de grandeza mais lento do que a implementação em C realizada neste trabalho.

Kawai, Garcia-Molina e Benjelloun, em seu trabalho [Kawai et al., 2006], estendem o conceito de *generic entity resolution* [Benjelloun et al., 2005] e implementam um algoritmo para pareamento de registros chamado *P-Swoosh*. Basicamente, essa família de algoritmos considera dois conjuntos de registros, R e R'. R contém os registros a serem pareados e R' será o conjunto-resultado. Os seguintes passos são realizados:

1. Escolher um registro de R como sendo o alvo, removendo-o de R.

2. Comparar o registro alvo com todos os registros de R'.

3. Se há concordância (casamento), mesclar todos os registros, gerando um novo com campos multi-dimensionais. Os registros que geraram esse novo são removidos. Esse novo registro é o canônico.

4. Se não há concordância, insere o registro alvo em R'.

5. Repetir o passo 1 até que R esteja vazio.

Parece claro não haver a etapa de blocagem. Contudo, esta etapa pode ser introduzida por meio de regras de usuário. Em um caso extremo, onde R não possui qualquer par verdadeiro, cada registro será comparado com outros $N * (N + 1)/2$ registros. O *P-Swoosh* pode funcionar com bases replicadas, isto é, cada processador tem sua própria cópia, ou em um esquema de *grid* onde os registros são divididos em b *buckets* disjuntos (diferentemente do FERAPARDA, que distribui pares de registros). Para esse esquema, $b * (b + 1)/2$ processadores são necessários. Cada processador executa o algoritmo de pareamento de registros sobre seu *bucket* e encaminha o resultado para um processador mestre. Após ter recebido todos os resultados, o processador mestre consolida a informação e distribui novamente os *buckets* até que o conjunto R esteja vazio. A versão paralela consegue um *speedup* praticamente linear até 15 processadores. Além disto, o trabalho tem uma discussão interessante sobre forma de paralelização e também sobre balanceamento de carga. Porém, sua implementação em Java e os testes realizados com bases de dados com 5 mil e 20 mil registros mostram que o *P-Swoosh* foi criado com o objetivo principal de testar os conceitos sem, efetivamente, possibilitar o processamento de grandes bases de dados reais. Por gerar uma grande quantidade de pares candidatos e agregar a cada momento novas informações ao registro canônico.

3.4 Sumário

Até onde pôde ser avaliado, nenhum trabalho na literatura discute maiores detalhes sobre a paralelização do problema de pareamento de registros. O assunto é amplamente estudado em suas etapas, como a blocagem e a comparação mas, geralmente, os experimentos consideram apenas dezenas de milhares de registros.

Capítulo 4

O Algoritmo Paralelo de Pareamento de Registros

Neste capítulo descrevemos a implementação do algoritmo de pareamento de registros em paralelo. A seção 4.1 apresenta o ambiente de execução Anthill, base fundamental para a construção e para a eficiência do algoritmo, discutindo os seus conceitos e abstrações fundamentais. A decomposição do processo de pareamento de registros em filtros é descrita na seção 4.2. O conjunto de premissas para a implementação do algoritmo e algumas decisões importantes são apresentadas e discutidas na seção 4.3.

4.1 Anthill

Esta seção apresenta o ambiente e o modelo de programação usados na implementação do algoritmo paralelo de pareamento de registros. O Anthill Ferreira et al. [2005] foi escolhido como ambiente de programação para a implementação. Ele permite que uma aplicação seja dividida em partes que podem ser instanciadas e executadas em diferentes unidades de processamento. Cada uma dessas partes realiza uma transformação sobre os dados e encaminha o resultado para a parte seguinte, formando uma espécie de *pipeline* de execução. Os autores do Anthill chamam esse conceito de modelo filtro-fluxo (*filter-stream model*).

No modelo filtro-fluxo, filtros são representações de cada estágio de computação onde os dados são transformados. Os fluxos são abstrações para a comunicação, permitindo a transferência de *buffers* de dados de tamanho fixo de um filtro para o próximo no *pipeline*, resumindo a criação de uma aplicação à decomposição em filtros.

Na decomposição em filtros, a aplicação é modelada como um fluxo de dados e então implementada como uma rede de filtros, sendo o paralelismo de tarefas realizado

por meio de um *pipeline*. Em tempo de execução, várias cópias de cada filtro da aplicação podem ser instanciadas em diferentes máquinas de um *cluster*. O ambiente Anthill permite dinamicamente conectar cada filtro origem ao destino por meio dos fluxos, como pode ser visto na Figura 4.1.

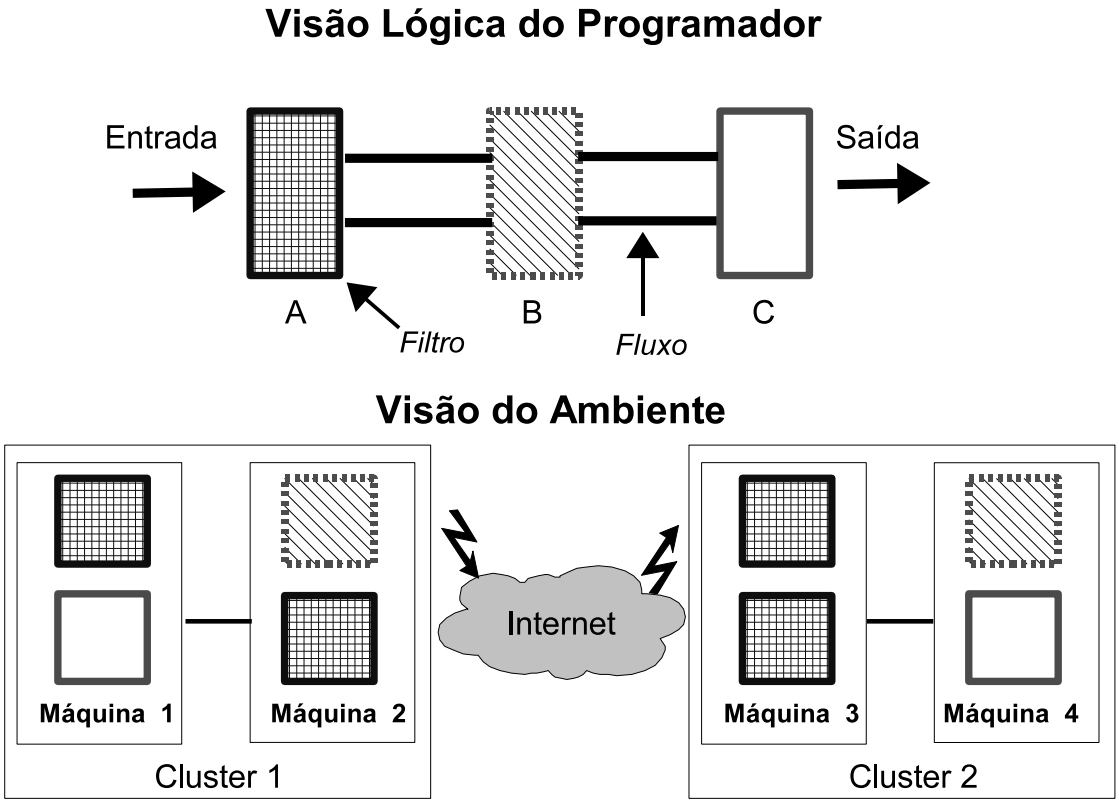


Figura 4.1. Abstração filtro-fluxo do Anthill

O Anthill explora três possibilidades de paralelismo: paralelismo de tarefa, de dados e assincronia. Ao dividir a computação em vários estágios de um *pipeline* (paralelismo de dados), cada estágio podendo ser replicado múltiplas vezes (para processar dados em paralelo), podemos ter um paralelismo de grão-fino e como tudo isto acontece de forma assíncrona, a execução poderá estar livre de gargalos.

Em várias aplicações construídas com o Anthill, foi possível observar que a solução natural freqüentemente era um grafo cíclico, onde a execução consistia de várias iterações sobre os filtros. A aplicação pode começar com uma representação inicial da solução e, a cada nova iteração pelo ciclo do *pipeline*, tal solução pode ser refinada. Esse comportamento leva a execuções assíncronas, pois poderão existir soluções (possivelmente de diferentes iterações), geradas simultaneamente em tempo de execução.

Em diversas aplicações, incluindo o processo de pareamento de registros, a computação apresenta uma natureza cíclica, ou seja, geralmente existem dependências entre

diferentes dados que trafegam pelo ciclo. Como cada estágio da computação pode ter várias réplicas, deve haver uma maneira de encaminhar o resultado de uma computação em dado ciclo de volta a uma instância específica em um ciclo posterior. Isto pode ser necessário, por exemplo, se existe algum estado associado com todas as partes interdependentes dos dados e uma delas reside apenas em uma instância de um filtro. Assim, todas as partes dependentes devem ser roteadas para tal instância específica. Para isto, o Anhill usa uma abstração chamada fluxo rotulado (*labeled stream*), característica que o difere de seus predecessores. Um fluxo rotulado é criado pelo programador para associar um rótulo a cada mensagem. Uma função (*hash*) também é definida para que cada rótulo aplicado a uma mensagem possa ser usado para encaminhar as mensagens para uma instância específica de um filtro (Figura 4.2).

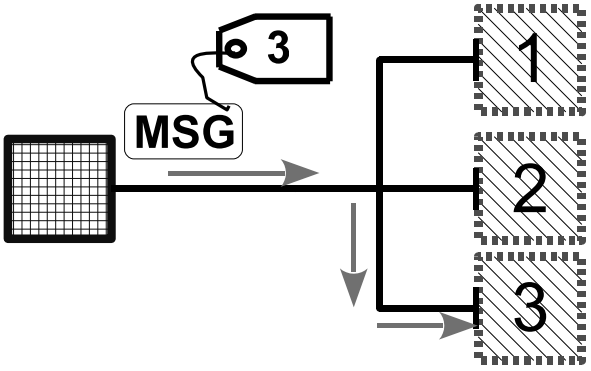


Figura 4.2. Uso de fluxos rotulados para especificar instância do filtro

O mecanismo de fluxo rotulado permite controle total da aplicação sobre o roteamento de suas mensagens. Como a função *hash* é chamada em tempo de execução, a decisão sobre o roteamento é tomada individualmente para cada mensagem e pode ser alterada dinamicamente durante a evolução da execução. Esta característica permite, entre outras coisas, a reconfiguração dinâmica para balanceamento de carga em aplicações irregulares. A função *hash* também pode ser relaxada de forma a permitir que a mensagem seja encaminhada para mais de uma instância (*multicast* ou *broadcast*). Isto é particularmente interessante para aplicações onde um único dado de entrada influencia vários dados de saída.

4.2 Paralelização do Algoritmo

A paralelização do processo de pareamento de registros segmentou a aplicação em seis filtros lógicos: *Reader*, *Blocking*, *Merger*, *Scheduler*, *Comparator* e *Classifier*. O mapeamento entre a etapa do processo e o filtro do Anthill que a realizada pode ser visto

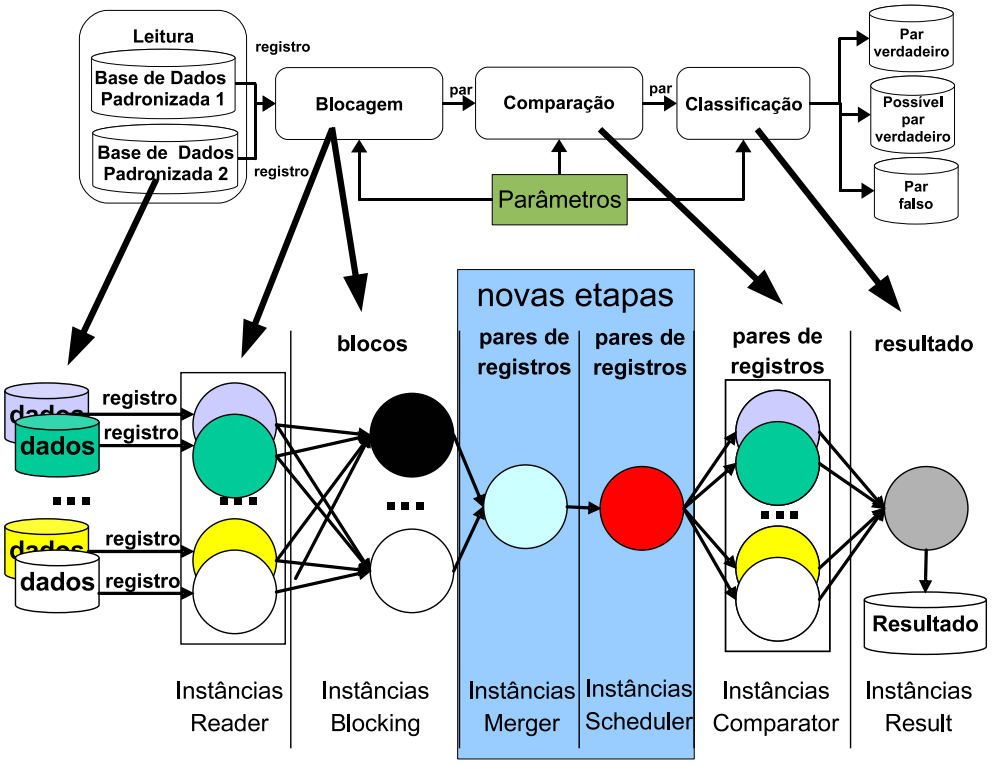


Figura 4.3. Pareamento de registro na visão de filtros lógicos

na Figura 4.3. Note que o existe uma relação direta entre a etapa de *leitura* e o filtro *Reader*, etapa de *blocagem* e o filtro *Blocking*, etapa de *comparação* e o filtro *Comparator* e etapa de *classificação* e o filtro *Result*. Note que a etapa de blocagem é realizada também no filtro *Reader* e duas novas etapas foram incluídas: uma para eliminar pares candidatos redundantes (filtro *Merger*) e outra para explorar a localidade de referência (filtro *Scheduler*). Nas subseções a seguir apresentaremos os as razões dessas decisões e os detalhes de cada filtro.

4.2.1 Filtro Reader

O filtro **Reader** (Algoritmo 3) é responsável por ler cada registro da base de dados, atribuir um identificador interno ao processo e gerar uma chave para conjunção do predicado de blocagem. A geração do identificador interno é feita utilizando-se um contador seqüencial para cada instância do filtro *Reader* (por meio da fórmula $id = totalDeInstancias * sequencia + rank$). Desta forma, podemos saber qual a origem de cada registro dentro do *pipeline*, viabilizando a utilização fluxos rotulados (*labeled-streams*), necessários em vários estágios. Note que a geração de chave de blocagem, algo que deveria ser feito na etapa seguinte, já ocorre neste momento. Com isto, somente

meta-dados são trafegados e não todo o registro.

```
1  Reader (Dataset, Configuration, Rank, Instances):
2  sequence ← 0
3  foreach Records ∈ Dataset do
4      Key ← ∅
5      Record.id ← sequence * Instances + rank
6      foreach Conjunction ∈ Predicate do
7          foreach Transformation ∈ Conjunction do
8              Key ← concatenate(Key, transform(Record))
9          sendToblocker(Record.id, Key, Conjunction)
10     sequence ← sequence + 1
```

Algoritmo 3: Algoritmo para o filtro *Reader*

O **balanceamento de carga** para este filtro é trivial. Cada instância ficará com uma partição de *n-avos* da base de dados (onde n é o número de instâncias). Cada instância gerará a mesma quantidade de chaves de blocagem e salvo diferenças mínimas, o custo será o mesmo.

4.2.2 Filtro Blocking

Assim que uma chave de blocagem é gerada, ela é enviada juntamente com o identificador do registro e identificador da conjunção para o filtro **Blocking** (Algoritmo 3, linha 9).

A comunicação entre o filtro *Reader* e o filtro *Blocking* é feita por meio de fluxos rotulados baseados na própria chave de blocagem. Não há como um mesmo bloco estar fragmentado entre as possíveis várias instâncias do filtro *Blocking*. Sabemos que, de acordo com a distribuição das chaves de blocagem, podem ocorrer problemas relacionados ao **desbalanceamento de carga**. Uma melhoria seria particionar o bloco entre as várias instâncias e enviar uma mensagem por *multicast* do filtro *Reader*, mas deixamos esse estudo como trabalho futuro.

O filtro *Blocking* (Algoritmo 4) mantém uma lista de todos os identificadores de registro que geraram a mesma chave de blocagem. Quando uma nova mensagem chega até esse filtro, ele identifica se é necessário criar um novo bloco ou simplesmente adicionar o registro a um já existente. Ainda durante a recepção da mensagem, o filtro *Blocking* gera os pares candidatos, como pode ser visto na Figura 2.2. Note que durante o processo de blocagem, os filtros de leitura continuam a gerar novas demandas, podendo levar à sobrecarga do *pipeline*. O Anthill é construído sobre o PVM [Sunderam, 1990] e não raramente encontramos problemas com o *buffer* de mensagens.

```
1  Blocking ():
2  HashTable ← ∅
3  foreach Message from Reader do
4      Key ← Message.key
5      Conjunction ← Message.conjunction
6      if Block ← get(HashTable, Key, Conjunction) == NULL then
7          Block ← createBlock()
8          put(HashTable, Key, Conjunction, Message.id)
9      foreach OldRec ∈ Block do
10         Pair ← sort(OldRec, Message.id) sendTomerger(Pair)
```

Algoritmo 4: Blocking filter algorithm

4.2.3 Filtro Merger

A Tabela 4.1 mostra como as disjunções de um predicado de blocagem podem se sobrepor e gerar pares candidatos repetidos, o que resultaria em processamento redundante. Os registros 1000 e 1100 são inseridos nos mesmos dois blocos (um para cada disjunção). Com isto, durante a geração de pares candidatos, o par $(1000, 1100)$ será gerado duas vezes.

Predicado = (sobrenome OU ano_nascimento)		
Identificador	**Nome**	**Nascimento**
1000	Luiz **Silva**	17/01/**1980**
1100	Felipe **Silva**	11/10/ **1980**

Tabela 4.1. Registros que geram pares candidatos redundantes para cláusulas de predicado

O filtro **Merger** (Algoritmo 5) eliminará quaisquer pares candidatos redundantes. Ele mantém um conjunto de todos os pares que foram gerados pelo processo de pareamento até o momento. Assim que um par redundante é identificado, ele é descartado. Uma otimização interessante aplicada neste filtro pode ser explorada pelo fato da geração de pares ser um processo crescente quando existe balanceamento de carga. Um par formado identificadores pequenos só ocorre no início e assim é possível descartá-lo pouco tempo depois. Isto permite manter um histórico de pares gerados pequeno, da ordem de dezenas de milhares, configurável através de parâmetro, implementado como uma lista circular.

Como o trabalho do filtro *Merger* é bem simples, não há necessidade de termos mais de uma instância.

```
1 Merger ():
2 HashTable ← ∅
3 foreach Message from Blocker do
4 |   Pair ← Message.pair
5 |   Key ← f(Pair)
6 |   if get(HashTable, Key) == NULL then
7 |   |   sendTocomparator(Message.pair)
8 |   |   put(HashTable, Key)
```

Algoritmo 5: Merger filter algorithm

4.2.4 Filtro Scheduler

O filtro *Scheduler* tem como objetivo organizar o fluxo de pares candidatos provenientes do filtro *Merger* de forma a diminuir o custo de comunicação (ver subseção 4.2.5), melhorar o balanceamento de carga distribuindo melhor os pares entre as instâncias do filtro *Comparator* ou um misto dos dois.

As mensagens geradas pelo filtro *Scheduler* precisam ser entregues a instâncias específicas do filtro *Comparator* e para isto são usados fluxos rotulados. A escolha da instância que receberá o par candidato para comparação considera a origem dos registros:

- **Ambos os registros que formam o par foram originados da mesma partição de dados**: O par é enviado para a instância que originou os registros. Não há comunicação entre instâncias do filtro *Comparator*.

- **Cada registro foi originado em uma partição de dados diferente**: Considerando um cenário onde não exista nenhum tipo de *cache* de comunicação, qualquer escolha é satisfatória, uma vez que todos os registros necessários deverão necessariamente ser comunicados. Entretanto, se utilizarmos uma *cache* de comunicação, a decisão deve utilizar alguma heurística a fim de maximizar sua utilização. Note que havendo ou não uma *cache*, o filtro *Scheduler* sempre envia o par para uma instância que tenha pelo menos um dos registros, pois, do contrário, o custo de comunicação dobraria.

Deixamos a discussão sobre o filtro *Scheduler* para o Capítulo 6, onde apresentamos as nossas abordagens para otimização do uso da *cache* e exploração da localidade de referência.

4.2.5 Filtro Comparator

Conceitualmente, os filtros *Reader* e *Comparator* são diferentes, mas, por questões de otimização (notadamente por causa da localidade de referência), eles são implementados como um único processo do sistema operacional (filtro *ReaderComparator*). Reforçamos que a leitura dos dados e a comparação dos registros continuam sendo duas tarefas diferentes e que a implementação como um único processo do sistema operacional foi decidida tendo como base a otimização e utilização de uma *cache*.

A Figura 4.4 mostra o *pipeline* em sua implementação real. Unir os filtros *Reader* e *Comparator* envolve criar um ciclo no *pipeline*. Como dito, no filtro *ReaderComparator* podemos ter dados em diferentes ciclos de processamento. Enquanto registros ainda são lidos, mensagens de comparação já são recebidas e processadas.

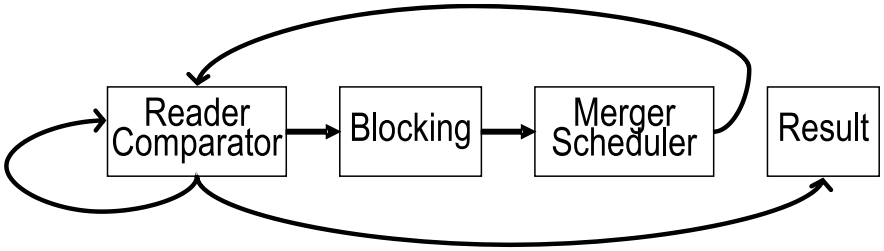


Figura 4.4. Visão de implementação dos filtros

O filtro *Comparator* sempre receberá um par a ser comparado onde pelo menos um dos registros está na partição local da base de dados (algoritmo 6, linha 8). Se ambos os registros estão presentes em sua partição dos dados (linha 11), o trabalho do filtro *Comparator* é simplesmente aplicar as regras de comparação e encaminhar o resultado para o próximo estágio do *pipeline* (linhas 12-14).

As instâncias do filtro *Comparator* podem receber três tipos de mensagens diferentes: *Compare (CmpMsg)*, *ReceiveAndCompareRecord (RCRMsg)* e *CompareAlreadyReceivedRecord (CRRMsg)*. Se uma instância de *Comparator* recebe uma mensagem *CmpMsg* e o registro complementar já se encontra na *cache*, nenhuma comunicação extra é realizada e a comparação é feita imediatamente (linhas 15-17).

Se um dos registros não faz parte da partição local, a instância verifica se o seu registro já foi enviado para a instância que possui o registro complementar. Em caso positivo, é enviada a mensagem *CRRMsg* (linhas 18-19), do contrário, envia-se a mensagem *RCRMsg* (linhas 20-22). Comparativamente, a mensagem *CRRMsg* é formada apenas pelos identificadores dos registros, ao passo que *RCRMsg* contém os mesmos identificadores e ainda todo o registro. Logo, diminuir a quantidade de mensagens *RCRMsg* reduz o custo de comunicação, uma vez que os registros podem

ocupar centenas de bytes. A instância que recebe o par e possui apenas um registro sempre o envia, dado que, com esta abordagem, diminuímos a comunicação e evitamos a necessidade de sincronização que existiria se cada instância do filtro *Comparator* tivesse que solicitar a outra instância o registro ausente.

Ao receber uma mensagem do tipo *CRRMsg*, basta a instância do filtro *Comparator* comparar o par. Note que assumimos que sempre que essa mensagem for recebida, existirá uma sincronia entre a *cache* e o emissor de forma que o último saberá quando um registro faz ou não parte da primeira. No caso de receber uma mensagem *RCRMsg*, a única diferença é que o registro é salvo na *cache* para uso futuro.

```
1  Comparator (DatasetPartition):
2  SentCache ← ∅
3  RecCache ← ∅
4  foreach Message received do
5      id1 ← Message.pair.id1
6      id2 ← Message.pair.id2
7      /*It Always has this record*/
8      Record1 ← get(DatasetPartition, id1)
9      if Message.type = Compare then
10         /*Instance has both records?*/
11         if id2 ∈ DatasetPartition then
12             Record2 ← get(DatasetPartition, id2)
13             R ← compare(Record1, Record2)
14             sendToclassifier(R)
15         else if id2 ∈ RecCache then
16             R ← compare(id1, id2)
17             sendToclassifier(R)
18         else if id2 ∈ SentCache then
19             sendToprocess(owner(id2), id1, id2)
20         else
21             sendRecord(owner(id2), record(id1), id2)
22             put(SentCache, id1, owner(id2))
23     else if Message.type = CRRmsg then
24         Record2 ← get(RecCache, id2)
25         R ← compare(Record1, Record2)
26         sendToclassifier(R)
27     else if Message.type = RCRmsg then
28         Record2 ← Message.Record
29         R ← compare(Record1, Record2)
30         sendToclassifier(R)
31         put(RecCache, id2)
```

Algoritmo 6: Algoritmo para o filtro *Comparator*

É importante lembrar que uma das premissas para o algoritmo é que a base de dados não está previamente ordenada. Não foi possível encontrar uma estratégia eficiente para particionar uma base de dados de forma a se obter um **balanceamento de carga** perfeito. Empiricamente, percebemos que não ocorrem grandes desbalanceamentos (ver Capítulos 5 e 7), mas não é possível generalizar.

A primeira versão do algoritmo considera uma *cache* de tamanho ilimitado para a comunicação de registros entre instâncias do filtro *Comparator*. Na prática, o tamanho pode ser limitado sem perda de desempenho (como será discutido no Capítulo 6). Por hora, consideramos que todos os registros comunicados entre as instâncias estarão na *cache* e nunca irão expirar. Assumimos uma abordagem conservadora e implementamos a *cache* individualizada por instância e mantemos uma sincronização entre o emissor e o receptor. Portanto, uma instância sabe exatamente quando outra possui determinado registro em sua *cache* de comunicação. Mesmo no caso de *cache* limitada, assumindo que a ordem das mensagens sempre é respeitada, torna-se possível manter a consistência.

4.2.6 Filtro Classifier

O último filtro é chamado de **Classifier** (Algoritmo 7). Ele classifica os pares de registros em concordância, discordância ou possível concordância (necessitando neste caso intervenção e análise humana), utilizando o pareamento probabilístico [Fellegi & Sunter, 1969].

```
1  Classifier ():
2  foreach Message from Comparator do
3      C ← Message.c
4      C' ← f(C)
5      if C' > upperthreshold then
6          Par é uma concordância.
7      else if C' < lowerthreshold then
8          Par é uma discordância.
9      else
10         Possível concordância.
```

Algoritmo 7: Algoritmo para o filtro *Classifier*

4.2.7 Extensões

Uma possível melhoria no processo de pareamento de registros seria a inclusão de um estágio que aplicasse a transitividade como discutida no trabalho de Hernandez [Her-

nandez & Stolfo, 1998]. Supondo que a blocagem não consiga cobrir todos os pares reais (registros acabaram em blocos diferentes), se houver pares capazes de ligar os diferentes blocos poderíamos aplicar a transitividade. Por exemplo, para um predicado de blocagem $P = \{C_1|C_2\}$ com duas conjunções, se um par verdadeiro $par_1 = (a, b)$ pertencer ao blocos gerados por C_1 e o par verdadeiro $par_2 = (b, c)$ pertencer ao bloco gerado por C_2, podemos assumir sem perda que o par $par_3 = (a, c)$ também é verdadeiro. Uma implementação paralela deste estágio é deixada como trabalho futuro.

4.3 Decisões de Implementação

A implementação do algoritmo de pareamento de registros em paralelo teve como base as seguintes premissas:

1. **A solução final deve se tornar um arcabouço onde novos filtros (abstrações do ambiente Anthill) poderão ser incluídos bem como algum filtro existente poderá ter sua implementação substituída.**

   Neste trabalho, apenas uma das técnicas de blocagem (a clássica) foi implementada. É interessante poder avaliar outras técnicas considerando-se o aspecto da paralelização e escalabilidade. A maior parte dos trabalhos sobre blocagem [Bilenko, 2006; Bilenko et al., 2006; Baxter et al., 2003] consideram apenas aspectos relacionados à cobertura de pares verdadeiros e total de pares gerados. Deixamos essa avaliação como trabalho futuro. A modelagem por meio de filtros e fluxos permite ainda imaginarmos outros estágios para o *pipeline*. Um exemplo é a inclusão de um filtro para aplicar a transitividade de pares verdadeiros [Hernandez & Stolfo, 1998] ou ainda incorporar aspectos semânticos ou algum refinamento da etapa de blocagem.

2. **A partição dos dados e sua disposição (ordenação) não são conhecidos de antemão.**

   A definição da melhor blocagem no processo de pareamento de registros muitas vezes é feita de forma empírica. A implementação não requer qualquer restrição sobre a disposição (ordenação dos dados). Desta forma, o usuário pode simplesmente trocar a definição da blocagem sem precisar reconstruir índices ou reordenar toda a base de dados. Evidentemente, podem existir situações onde o algoritmo teria ganhos ao utilizar dados dispostos de forma a explorar ao máximo a localidade de referência. Contudo, não encontramos uma forma de tornar

isto válido para todos os casos. A ordenação também poderá provocar um desbalanceamento de carga. Assumindo que toda a base é ordenada e depois particionada entre as instâncias, dependendo do predicado de blocagem poderemos ter um bloco muito grande ficando a cargo de apenas uma instância de filtro. É o caso de atributos mais freqüentes.

3. **Não é necessário processar todos os possíveis pares candidatos, mas é fundamental cobrir a maior parte dos pares reais.**

   A etapa de blocagem é crítica para o processo. Quanto menos restritiva, maior o número de pares candidatos gerados, causando uma explosão combinatória. Por outro lado, com uma blocagem restritiva, pares reais poderão excluídos do bloco formado. Sabendo deste compromisso, preferimos construir uma solução que seja robusta o suficiente para processar grandes quantidades de pares e que explore aspectos da localidade de referência para maximizar o *throughput*.

4. **Por maior que seja o número de máquinas no *cluster*, ainda assim poderá existir uma base de dados que não caberá em memória principal.**

   A implementação deve trabalhar com um conjunto de dados pequenos em memória principal, descartando os dados após o processamento e tendo como recuperá-los da memória secundária quando necessário. Foi implementada uma *cache* que armazena em memória principal os registros da partição de dados local a cada instância do filtro *Reader*. Também existem abstrações na implementação que permitem estender o conceito de fonte de dados. Uma fonte de dados pode ser um arquivo, uma conexão a um servidor de banco de dados ou qualquer outra fonte que forneça dados estruturados.

5. **Existe a possibilidade de que haja um desbalanceamento provocado pela irregularidade dos dados.**

   Um filtro escalonador (*scheduler*) deve ser definido para rotear os pares candidatos para uma das instâncias do filtro *Comparator*. Sua implementação tem como objetivo diminuir o número de mensagens trocadas entre os filtros (explorando a localidade de referência), servir como um balanceador de carga ou algo um misto de ambos.

6. **A solução é livre e aberta.**

   Gostaríamos que a solução contribuísse com outros projetos de pesquisa, pudesse ser estendida e também fosse usada como ferramenta por empresas e pelo Gov-

erno. No estágio em que se encontra, a ferramenta necessita de um especialista para a configuração de parâmetros, análise dos resultados, ajustes finos e gestão de bases de dados. Deixaremos como trabalho futuro a melhoria da *interface* de usuário.

4.4 Discussão

Consideramos que a implementação do nosso algoritmo escala por explorar alguns aspectos do paralelismo de tarefas, paralelismo de dados e uso da assincronia, disponíveis no ambiente de execução Anthill.

O paralelismo de tarefas é implementado por meio de um pipeline controlado pela dependência dos dados, como apresentado na Figura 4.3. As instâncias do filtro *Reader* não precisam ler toda a partição de dados antes de enviar a informação para o filtro *Blocking*. Assim que um registro é lido, ele é imediatamente colocado no pipeline para processamento. Como resultado, múltiplas tarefas podem ser executadas simultaneamente.

Ao utilizar mais de uma instância dos filtros, estamos aproveitando o paralelismo de dados. Notadamente, os filtros *Reader* e *Comparator* exploram bem essa característica. Outro filtro que também é bom candidato ao paralelismo de dados é o *Blocking*. As instâncias desse filtro iteram sobre uma lista de identificadores de registros em $O(n)$, sendo n o tamanho médio dos blocos. Quando usamos muitas instâncias do filtro *Reader* ou mesmo quando tamanho médio n é grande, o filtro *Blocking* pode ficar sobrecarregado. Neste caso, podemos utilizar mais de uma instância, apesar de que a distribuição das chaves de blocagem pode provocar um desbalanceamento de carga.

A assincronicidade é explorada primeiramente por meio da eliminação de pares redundantes no filtro *Merger* (portanto, não há relação 1 : 1 entre pares candidatos gerados e efetivamente comparados). Ainda podemos executar mais de um instância de um filtro ou mesmo filtros diferentes em um mesmo computador (especialmente se multi-core). Enquanto um processo está ocioso, outro pode fazer uso dos recursos. Observamos que, por meio do uso de multiprogramação, o desempenho do algoritmo pode ser otimizado, especialmente em relação ao uso de CPU.

A sobrecarga do *pipeline* em certas ocasiões é um problema que pode levar a um consumo excessivo de memória, devido aos *buffers* de comunicação do PVM. Idealmente, deveria ser implementado um mecanismo de sincronização entre produtor e consumidor. Esta questão fica em aberto e possivelmente será resolvida com a nova versão do Anthill que usa como base o MPI [Gropp et al., 1996].

Capítulo 5

Avaliação do Algoritmo

Neste capítulo apresentamos a avaliação experimental do nosso algoritmo paralelo de pareamento de registros. Aqui consideramos a primeira versão do algoritmo, sem otimizações relacionadas à localidade de referência.

5.1 Experimentos

Os experimentos foram executados em um cluster formado por computadores AMD Athlon 64 3200+ com 2GB de RAM, conectados por uma rede Gigabit Ethernet e executando o sistema operacional Linux 2.6.

5.1.1 Caracterização das Bases de Dados

Dada a dificuldade de se conseguir bases de dados reais grandes o suficiente para a realização dos experimentos, decidimos utilizar o gerador sintético de carga *DsGen* provido pela ferramenta Febrl [Peter Christen, 2004]. O gerador reproduz certas características dos dados reais, como ausência, erros tipográficos, variações fonéticas, freqüência e variações na escrita de nomes e mesmo variações em endereços postais. O DsGen possui uma série de tabelas que permitem-no gerar bases bem próximas às reais, com uma consideração: a base segue características do idioma inglês. Como parâmetros de entrada, o DsGen espera o nome de uma função de distribuição de probabilidade, a quantidade de erros introduzidos por registro e a quantidade de réplicas. Os registros gerados pelo DsGen têm tamanho variável e são formados pelos atributos descritos na tabela A.1 do Apêndice A.

Foram geradas bases de até 1 milhão de registros, com 10% sendo réplicas e, em média, 5 erros introduzidos por registro. Utilizamos a distribuição de probabilidade

uniforme, conforme exemplos do Febrl. Entendemos que outras distribuições poderiam ser utilizadas, inclusive impactando na forma como a carga é distribuída (mais réplicas geram mais pares candidatos e conseqüentemente, mais processamento). Entretanto, deixamos a análise para trabalhos futuros. Os valores utilizados foram selecionados de forma que pudéssemos comparar os resultados com aqueles gerados pelo Febrl em seu projeto que consideramos a *linha de base*.

5.1.2 Avaliação dos Resultado

As réplicas geradas pelo DsGen possuem uma referência ao registro original, permitindo assim avaliar a eficiência do algoritmo. Na nossa avaliação final, mais de 95% dos pares reais foram corretamente classificados. Creditamos essa taxa alta de acerto principalmente à grande quantidade de pares gerados, mais de 270 milhões para 1 milhão de registros, que na prática poderia ser bem menor. Os pares verdadeiros que não foram corretamente identificados situaram-se na região entre os dois limiares e portanto precisariam de intervenção manual. Seguimos os mesmos parâmetros para os limiares $T_{superior}$ e $T_{inferior}$ usados pelo Febrl: 30,0 e 0,0 respectivamente.

5.1.3 Definição dos Parâmetros para os Experimentos

Além da base de dados, os principais parâmetros para o processo de pareamento de registros são as definições da blocagem, os critérios de comparação e classificação.

Para avaliar a escalabilidade, decidimos utilizar predicados de blocagem que gerassem uma grande quantidade de pares candidatos. Em uma situação normal, um predicado mais restritivo atenderia perfeitamente. No máximo, foram gerados 270 milhões de pares em uma base de 1 milhão de registros (270 comparações por registro).

$$
\begin{aligned}
P \;=\; & (phone\_number \wedge first\_name) \;\cup \\
& (phone\_number \wedge year\_month\_birthdate) \;\cup \\
& (dmetaphone(last\_name, 4) \wedge year\_birthday) \;\cup \\
& (dmetaphone(first\_name, 4) \wedge birthdate) \;\cup \\
& (zip\_code \wedge birthdate) \;\cup \\
& (truncate(first\_name, 3) \wedge zip\_code) \;\cup \\
& (nysiis(locality, 4) \wedge month\_birthdate)
\end{aligned}
$$

Os critérios de comparação usados estão listados no Apêndice B. Usamos funções de comparação de caracteres como Jaro [Jaro, 1989], uma vez que, além de possibilitarem obter um resultado melhor do que a comparação exata, também são computacionalmente intensivas e assim foi possível avaliar melhor a escalabilidade do filtro *Comparator*.

5.1.4 Avaliação da Escalabilidade

A análise preliminar do algoritmo seqüencial utilizando a ferramenta `gprof` mostrou que mais de 90% do tempo de execução é gasto nas comparações. Por esta razão, baseamos nossa análise de *speedup* apenas no número de instâncias do filtro *Comparator*.

Realizamos um experimento variando a quantidade de instâncias dos filtros *Reader* e *Comparator*. Como dito, o filtro *Reader* e o filtro *Comparator* são um único processo. Cada instância destes filtros foi executada em um computador diferente, ficando os demais filtros (*Blocking*, *Merger*, *Scheduler* e *Result*) em um único computador para até 8 instâncias de *Comparator* e em 2 computadores para mais 8 instâncias.

A Figura 5.1 mostra os resultados do experimento quando variamos o número de registros em cada base de dados e o número de instâncias dos filtros *Reader* e *Comparator*. Usamos bases de dados com 250.000, 500.000 e 1 milhão de registros. Analisando o gráfico de *speedup*, podemos perceber que o *speedup* segue o mesmo padrão para as três bases de dados utilizadas, alcançando uma eficiência de mais de 80% com 15 instâncias. Acreditamos que esse resultado é devido às características do algoritmo e do ambiente onde ele é executado (que permite explorar diferentes aspectos da paralelização como discutido na Seção 4.4).

Quando aumentamos o número de instâncias do filtro *Comparator*, a demanda por comunicação entre as instâncias aumenta, uma vez que a chance de encontrar um registro em uma das partições é diretamente proporcional ao tamanho dessa partição.

A Figure 5.2 mostra o aumento do número de mensagens trocadas em função do número de instâncias dos filtros *Comparator*. O gráfico foi construído utilizando-se uma base de dados de 250 mil registros, totalizando cerca de 1 milhão de pares. Analisando-o, podemos ver que o número de mensagens trocadas com 15 instâncias dos filtros é quase duas vezes o número de mensagens trocadas com uma única instância. Um total de 2,2 milhões de mensagens são trocadas entre os diferentes filtros independentemente da configuração (são intrínsecas do algoritmo). Para 15 instâncias, mais 1,5 milhão de mensagens adicionais são necessárias para comunicação apenas entre instâncias do filtro *Comparator*.

Como já foi dito, a utilização da *cache* proposta reduziu significativamente o custo

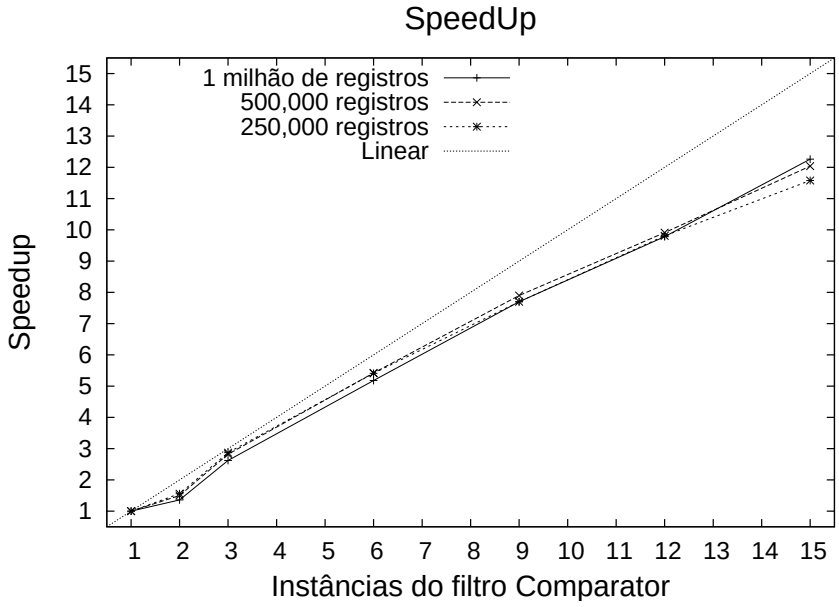


Figura 5.1. Avaliação de *speedup*

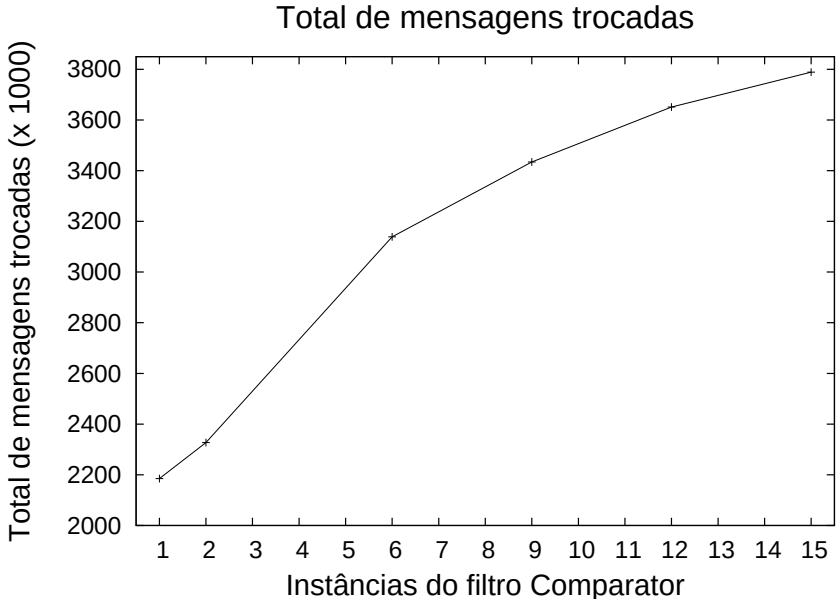


Figura 5.2. Mensagens trocadas entre filtros

a comunicação entre instâncias do filtro e *Comparator*. Os registros são enviados apenas uma vez e mantidos na *cache*. Para os pares subseqüentes, formados pelos registros já enviados, temos que somente os identificadores são enviados, reduzindo assim o custo de comunicação. Além disto, nós agrupamos vários registros e identificadores antes de enviá-los, a fim de reduzir ainda mais a sobrecarga causada pela comunicação.

A Figura 5.3 mostra o número absoluto de comparações por segundo ao longo de toda a execução de um experimento. Neste experimento usamos 3 instâncias do filtro *Comparator*. Podemos ver que mais de 60% das comparações (ou aproximadamente 30000 comparações por segundo) foram feitas utilizando apenas o identificador do registro (mensagem do tipo *CompareAlreadyReceivedRecord*). Outros 40% das comparações (algo em torno de 15000 comparações por segundo) foram feitas utilizando-se registros da partição local da base. Também podemos ver que o algoritmo tem um tempo inicial de aquecimento durante o qual existe uma alta taxa de troca de registros. Mas, após certo tempo, a troca de registros cai para praticamente zero.

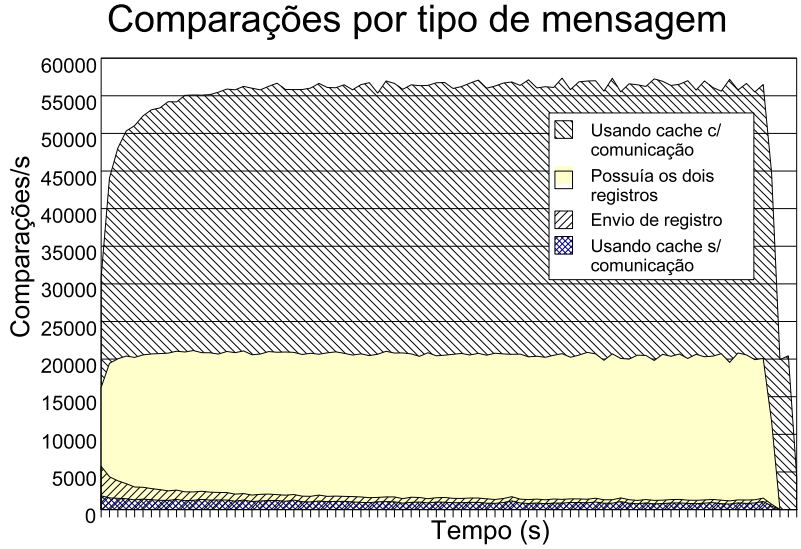


Figura 5.3. Quantidade de cada tipo de mensagem que originou comparação de registro

O gráfico percentual de comparações realizadas decorrentes de cada tipo de mensagem ao longo da execução do experimento é mostrado na Figura 5.3. Cerca de 4% das comparações são feitas utilizando a *cache* sem comunicação de meta-dados (identificadores de registros e de origem). Este tipo de comparação ocorre quando um par é enviado a uma instância do filtro *Comparator* e ele possui um dos registros e já armazenou o complementar em *cache* em algum tempo passado.

Para avaliar a eficiência da *cache*, coletamos estatísticas de sua utilização. Essas estatísticas são apresentadas na Tabela 5.1. Podemos ver que o percentual de comparações realizadas utilizando-se a *cache* aumenta com o número de instâncias do filtro *Comparator*. Quanto mais instâncias, menor é a partição local dos dados e portanto maior a necessidade de comunicação e utilização da *cache*.

Outra razão para o *speedup* observado é o balanceamento de carga nas instâncias

# Inst.	**T**otal de comparações utilizando *cache*	%
1	0	0
2	134454100	49.68
3	178538727	65.95
6	221754408	81.78
9	235108278	86.63
12	240764988	88.70
15	243796260	89.77

Tabela 5.1. Estatísticas das comparações para 1 milhão de registros

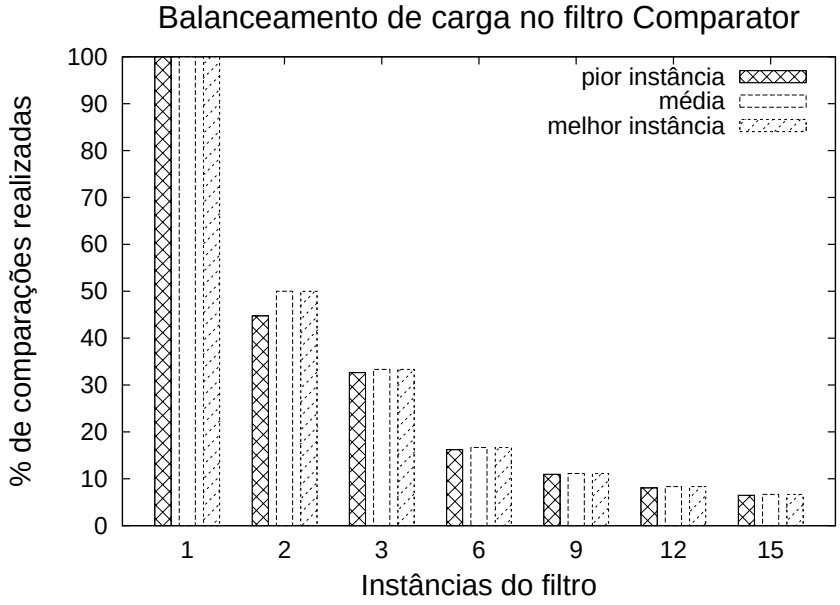


Figura 5.4. Balanceamento de carga considerando pares comparados

do filtro *Comparator*. O gráfico da Figura 5.4 mostra o percentual do total de comparações realizadas pela pior instância (realizou menos comparações), melhor instância (realizou o maior número de comparações) e a média de comparações de todas as instâncias. A diferença entre a pior e a melhor instância se torna muito pequena quando o número de instâncias cresce, mantendo um balanceamento de carga muito bom.

Os experimentos foram executados de forma que não fosse necessário utilizar a memória virtual das máquinas. O experimento com 1 milhão de registros e 270 milhões de pares coube em memória, mas utilizou quase 100% desta.

Poderia ser questionada a necessidade de um *pipeline*, uma vez que 90% do tempo de processamento se encontra em apenas um dos estágios. Vale ressaltar que sem o *pipeline*, a paralelização da comparação não seria simplesmente particionar os dados. Existe uma dependência não-trivial dos registros que formam os pares. Além disto, graças à modelagem como um *pipeline*, é possível adicionar novos estágios. Por fim,

existiriam cenários onde o tempo de processamento não ficaria tão concentrado na comparação. Um exemplo disto é a utilização de blocagens mais elaboradas, como proposto por Bilenko et al. [2006], quando um algoritmo baseado no algoritmo de cobertura de vértices é utilizado para obter maior precisão e cobertura.

No próximo capítulo discutiremos como diminuir a demanda de memória explorando a localidade de referência.

Capítulo 6

Entendendo e Explorando a Localidade de Referência

Como visto no capítulo 5, o uso da *cache* de registros entre instâncias dos filtros foi fundamental para alcançar a escalabilidade. Durante a primeira parte deste trabalho, tínhamos apenas noção dos motivos que levaram a esse fato, mas seu entendimento não estava completo. Neste capítulo, discutiremos como a localidade de referência influenciou diretamente os resultados obtidos. Mostraremos que a blocagem tem papel fundamental na forma como os pares candidatos são gerados e comparados e por isto impacta diretamente a localidade de referência temporal.

6.1 Localidade de Referência

O princípio da localidade de referência tem papel importante no projeto de sistemas de computação e sua aplicação é bem conhecida [Denning & Schwartz, 1971]. Consideramos dois tipos de localidade de referência: a temporal e a espacial. Na localidade de referência temporal temos um recurso que foi referenciado em um certo momento do tempo será referenciado novamente em um futuro breve. Já na localidade de referência espacial, temos que a possibilidade de referenciar um recurso é maior se outro recurso próximo a ele foi recentemente referenciado.

Consideramos que uma seqüência de acessos aos dados no processo de pareamento de registros apresenta *localidade temporal* se um registro recentemente acessado é novamente referenciado logo em seguida. Isto acontece sempre que dois ou mais pares candidatos possuem um mesmo registro em comum e são gerados em um certo intervalo de tempo.

A *localidade espacial* ocorre quando a possibilidade de referenciar uma região de

armazenamento é maior se a região próxima foi recentemente referenciada. Podemos explorar a localidade espacial durante a etapa de comparação. Considere um cenário onde temos duas instâncias do filtro *Comparator*, $Comparator_0$ e $Comparator_1$, cada um com uma partição de metade do total de registros. Considere um conjunto de pares de registros $C = \{(r_a, r_x), (r_b, r_x), \ldots, (r_n, r_x)\}$, onde r_x pertence a $Comparator_0$ e todos os outros registros pertencem a $Comparator_1$. Como são instâncias diferentes do mesmo filtro, existe a necessidade de comunicação entre eles. Enviar r_x para $Comparator_1$ explora melhor a localidade espacial e reduz os custos de comunicação relativamente a copiar todos os outros registros de $Comparator_1$ para $Comparator_0$

Os principais objetivos do estudo da localidade de referência são:

- Implementar uma *cache* para os registros da partição local, de forma que apenas um pequeno subconjunto esteja na memória principal a cada momento.
- Implementar uma *cache* de troca de registros entre instâncias dos filtros de forma a reduzir a comunicação.
- Entender quais fatores são importantes ao definir políticas de escalonamento dos pares candidatos.
- Entender como a etapa de blocagem influencia a localidade de referência.
- Alterar a implementação de forma que bases de dados ainda maiores possam ser utilizadas.

6.2 Evidência da Localidade Temporal

Para entender melhor a localidade temporal e o padrão de acesso aos registros da *cache* de registros implementada, definimos um modelo de distância de pilha (*stack distance model* [Mattson et al., 1970; Spirn, 1976]. Registros freqüentemente referenciados resultam em distâncias de pilha menores. Neste caso, esperamos que a taxa de acertos (*hits*) cresça rapidamente quando aumentamos a distância de pilha.

O padrão de acesso à *cache* é formado por uma seqüência de registros sendo solicitados pelas instâncias do filtro *Comparator*. Tal seqüência é definida logo na etapa de blocagem, assim que um novo par candidato é descoberto, podendo ser alterada no filtro *Scheduler*.

Para mostrar a presença da localidade temporal, gravamos um *trace* com o padrão de acesso à *cache* e geramos o gráfico da distância de pilha cumulativa quando a seqüência registrada no *trace* é aleatoriamente alterada e o comparamos com o gráfico

da distância de pilha cumulada do *trace* original [Almeida et al., 1996]. O novo *trace* mantém a popularidade dos registros, mas qualquer localidade temporal ou espacial que porventura existisse é perdida.

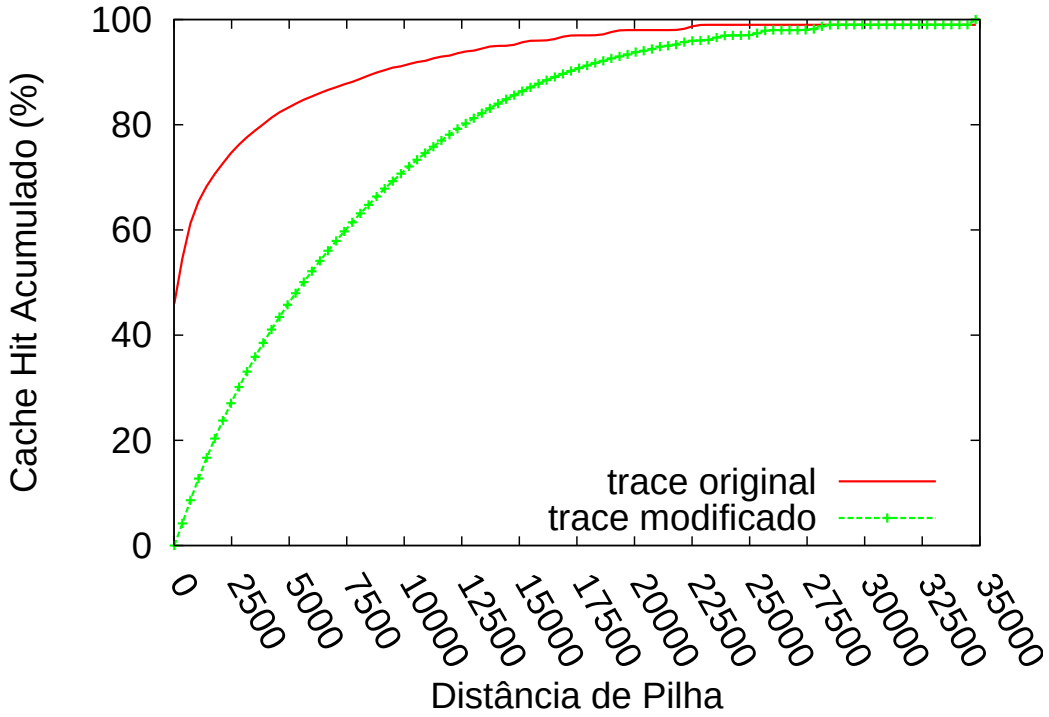


Figura 6.1. Distância de pilha para os *traces* original e modificado

Na Figura 6.1 observamos o gráfico da distância de pilha cumulativa para um experimento utilizando-se 35000 registros e duas instâncias dos filtros *Comparator*. Neste experimento, gravamos um *trace* do padrão de acesso à *cache* de registros e calculamos a distância de pilha. O eixo y representa o percentual de acessos à *cache* que resultaram em um *cache hit* e o eixo x representa o tamanho da *cache*. Note que com uma *cache* de aproximadamente 350 registros registros (1% do total), a taxa de *hits* foi de 54% com o *trace* original e apenas 4% com o *trace* modificado. Com 7000 registros (20%), no *trace* original a taxa de *hits* atinge 87% e no modificado, 58%.

Uma hipótese é que o tamanho da *cache* poderá ser reduzido, significando que o algoritmo paralelo consegue trabalhar, a cada momento do tempo, com um pequeno conjunto de registros sem perda considerável de desempenho.

6.3 Explorando a Localidade de Referência Temporal

A utilização de memória sempre foi uma das preocupações da construção do algoritmo de pareamento de registros, uma vez que um de seus requisitos é a capacidade de processar grandes quantidades de registros. Como dito no capítulo 4, a primeira implementação tratou todos os registros em memória principal, tanto os da partição local, quanto os registros que foram recebidos. Porém, não são poucos os casos onde a base de dados não cabe inteiramente em memória, mesmo em um *cluster*. Manter toda a partição da base de dados em memória secundária gera uma sobrecarga, uma vez que não se pode determinar um padrão de acesso aos registros (acesso é aleatório).

Uma solução intermediária é utilizar uma *cache* de registros na própria aplicação. Como discutido na seção 6.2, aparentemente existe um conjunto pequeno de registros que durante um intervalo de tempo é freqüentemente referenciado.

A fim de explorar a localidade de referência temporal, implementamos duas *caches*, uma para a partição local da base de dados e outra para os registros que são comunicados entre as instâncias do filtro *Comparator*. A primeira é chamada de *cache local* e a segunda de *cache de comunicação*.

6.3.1 Utilizando a Cache de Registros da Partição Local

Durante o processo de leitura dos registros e geração das chaves de blocagem (etapa que ocorre no filtro *Reader*), os registros da partição local da base de dados são lidos de forma seqüencial e gravados na *cache local*. Entretanto, mesmo antes de toda a partição da base de dados ter sido completamente lida, pares de registros já são recebidos no filtro *Comparator* (lembramos que os filtros *Reader* e *Comparator* são um só processo do sistema operacional), fazendo com que o padrão seqüencial de acesso aos registros seja quebrado. A *cache* de registros da partição local permite que o acesso aos registros seja mantido de forma seqüencial e com isto, outros níveis de *cache* (disco, sistema de arquivos, *caches* de bancos de dados, entre outros) sejam explorados. Além disso, a *cache* de registros da partição local permite que se reduza o tempo de acesso aos registros quando este for longo, por exemplo, quando o acesso é feito a um servidor de banco de dados em outro computador da rede.

Utilizamos a política LRU para a *cache* de registros da partição local, mas não avaliamos se outras políticas seriam melhores. Como mostraram os experimentos com a *cache* de registros da partição local (Capítulo 7), quando usamos registros armazenados em arquivos locais o tamanho desta cache pode ser bem pequeno. Preferimos deixar a análise mais detalhada sobre essa *cache* para trabalhos futuros.

6.3.2 Reduzindo a Comunicação Através da Localidade de Referência

A comunicação de um registro entre instâncias do filtro *Comparator* é a que transfere a maior mensagem do algoritmo. Dependendo da base de dados, o tamanho dessa mensagem pode ser de dezenas até centenas de *bytes*. A primeira implementação do algoritmo já reduzia a comunicação utilizando uma *cache* de comunicação. Sempre que um registro era enviado de uma instância do filtro para outra, ele era inserido na *cache* e nunca mais transferido (entre essas duas instâncias). Como não havia uma política para reposição da *cache* de comunicação, a implementação poderia ter, dependendo dos parâmetros da blocagem, toda a base de dados armazenada na memória de todas as instâncias do filtro *Comparator*.

```
1 EnviarPar (par, instancias):
2 if PertenceMesmaInstancia(par.registro1, par.registro2) then
3     destino ← ObterInstancia(par.registro1)
4 else
5     destino ← EscolherInstancia(par)
6 EnviaParParaComparacao(destino, par)
```

Algoritmo 8: Algoritmo escolha da instância que receberá o par a ser comparado

Como não é factível manter sempre toda a base de dados em memória, foi preciso definir uma política de reposição e um tamanho limitado para a *cache* de comunicação. Para entendermos como podemos explorar a localidade de referência a fim otimizar o uso da *cache*, é importante entender quando e por quê ela ocorre. Quando um par candidato é descoberto na etapa de blocagem, o par será encaminhado para a comparação. Na implementação, isto significa dizer que existe a decisão sobre qual instância do filtro *Comparator* irá receber o par. A decisão é simples quando ambos os registros do par candidato pertencem a uma mesma instância. Porém, se cada registro pertence a instâncias diferentes, a cada momento toma-se uma decisão que, desconsiderando-se qualquer tipo de *memória*, tem a probabilidade de 50% de ser a melhor. Se considerarmos algum tipo de *memória*, isto é, manter a informação sobre o destino dos pares anteriores, podemos diminuir a comunicação explorando a localidade de referência. Resumidamente, podemos descrever a decisão sobre qual instância receberá o par a ser comparado usando o algoritmo 8. Para as funções EscolherInstancia() e EnviarPar(), propomos a heurística a seguir.

6.3.2.1 Heurística

Para implementarmos a função `EscolherInstancia()`, definimos uma heurística. Esta heurística considera um modelo de filas de saída no filtro *Scheduler*. Para o conjunto $T = \{t_0, t_1, \ldots, t_n\}$ formado por t instâncias-destino do filtro *Comparator*, existirá um conjunto $Q = T \times T$ de filas, ou seja, cada combinação 2 a 2 com repetição possível de destino será formada. Por exemplo, se temos 3 instâncias do filtro *Comparator*, teremos as filas $Q = \{(t_0, t_0), (t_0, t_1), (t_0, t_2), (t_1, t_1), (t_1, t_2), (t_2, t_2)\}$. Não estamos interessados em quando ambos os pares pertencem à mesma instância (nos casos (t_0, t_0), (t_1, t_1) e (t_2, t_2)), pois a escolha é bastante clara. A idéia é que cada par gerado seja encaminhado para uma das filas de saída e cada fila de saída seja otimizada de forma a explorar o máximo a localidade de referência.

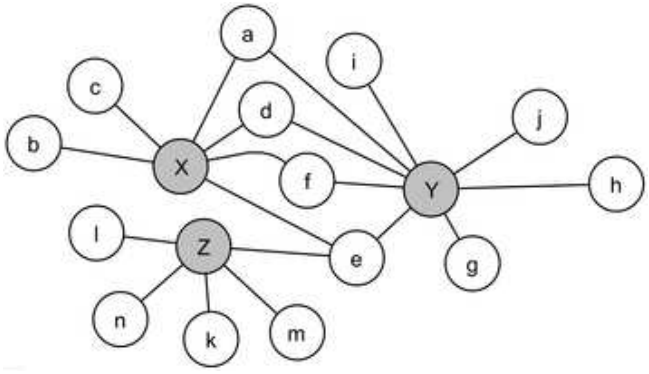


Figura 6.2. Referência espacial em um grafo representando pares candidatos

Considere a Figura 6.2. Se o grafo representa uma das filas de saída do filtro *Scheduler*, e os vértices brancos e cinzas pertencem a instâncias diferentes do filtro *Comparator*, a quantidade de registros a serem comunicados é menor se copiarmos os vértices em cinza. O algoritmo 9 mostra uma heurística para a função `EnviarPar()`. A heurística agrupa os pares a serem comparados em um grafo (linha 6). Os vértices deste grafo representam os registros e as arestas representam uma relação de pareamento. Quando o tamanho máximo do grafo é atingido (linha 8), recupera-se o vértice de maior grau (linha 10). O destino de todas as mensagens de comparação será a instância que contem o vértice de maior grau (linha 11). Cada aresta desse vértice é recuperada para reconstruir um par para comparação (linhas 12-15) e finalmente o par é enviado para comparação (linha 16) e o vértice é removido, juntamente com todas as suas arestas (linha 17). O envio de pares para a comparação continua até que não haja mais arestas no grafo.

Uma dúvida que pode surgir é a razão de sempre enviar os pares para a instância que possui vértice de maior grau. A razão desta escolha é que, para um par a ser com-

parado, somente uma mensagem é enviada entre as instâncias do filtro *Comparator*. O primeiro par que for formado pelo registro de maior grau implicará necessariamente uma mensagem do tipo *ReceiveAndCompareRecord (RCRMsg)*. Para os próximos pares, será usada somente a mensagem *CompareAlreadyReceivedRecord (CRRMsg)*. Concentrando o envio dos pares para a instância que possui o vértice de maior grau permite que esta instância controle o tipo de mensagem a ser enviada.

```
1  EnviarPar (par, limite, grafo):
2  if PertenceMesmaInstancia(par.registro1, par.registro2) then
3      destino ← ObterInstancia(par.registro1)
4      EnviaParParaComparacao(destino, par)
5  else
6      AdicionarAoGrafo(grafo, par.registro1, par.registro2)
7      /*Tamanho máximo do grafo atingido?*/
8      if Tamanho(grafo) > limite then
9          while Tamanho(grafo) > 0 do
10             vertice1 ← ObterVerticeMaiorGrau(grafo)
11             destino ← ObterInstancia(par.registro1)
12             foreach aresta ∈ Arestas(vertice1) do
13                 vertice2 ← Terminal(grafo, vertice1, aresta)
14                 novoPar.registro1 ← vertice1
15                 novoPar.registro2 ← vertice2
16                 EnviaParParaComparacao(destino, par)
17             Remover (grafo, node1)
```

Algoritmo 9: Função EnviarPar() modificada para a heurística

Apesar de simples, o algoritmo reduz consideravelmente a quantidade de registros comunicados quando é aplicado sobre um grafo denso. Entretanto, o custo de se manter o grafo atualizado consome tempo e memória quando consideramos grafos com um número muito grande de vértices. Quanto mais eficiente for a heurística, menos registros serão comunicados, resultando em um uso mais eficiente da *cache*. Como visto no capítulo 5, mais de 50% das comparações, podendo chegar até próximo de 90%, são feitas com o uso da *cache*. Com isto, a sobrecarga do processamento introduzida pela heurística aplicada a todos os pares faz com que ela ainda não seja a mais eficiente quando consideramos o tempo de execução (como será visto a seguir).

6.3.3 Influência da Blocagem na Localidade de Referência

Vamos rever o algoritmo 10 que realiza a etapa de blocagem (técnica clássica) para entender como são gerados os pares. Assim que uma mensagem contendo o registro e a chave de blocagem provenientes do filtro *Reader* chegam ao filtro *Blocking*, a chave de

blocagem é usada para recuperar da lista invertida todos os identificadores de registros recebidos anteriormente. Cada registro contido na lista invertida formará um par candidato com o registro recém-chegado ao filtro *Blocking*. Modelando o problema como feito na heurística apresentada anteriormente, porém de forma local, teríamos um grafo com $(n+1)$ vértices (n é a quantidade de registros recuperados da lista invertida), sendo que o registro recém-chegado seria o vértice com maior grau (n). Considerando uma aplicação local da heurística proposta, o registro recém-chegado é o melhor candidato a ser copiado para outra instância, pois resulta no menor número de registros sendo comunicados.

```
1  Blocking ():
2  HashTable ← ∅
3  foreach Message from Reader do
4      Key ← Message.key
5      Conjunction ← Message.conjunction
6      if Block ← get(HashTable, Key, Conjunction) == NULL then
7          Block ← createBlock()
8          put(HashTable, Key, Conjunction, Message.id)
9      foreach OldRec ∈ Block do
10         Pair ← sort(OldRec, Message.id) sendTomerger(Pair)
```

Algoritmo 10: Algoritmo do Filtro *Blocking*

Uma maneira simples e eficiente de descobrir o vértice com maior grau para a solução local surge quando revemos a forma como os identificadores de registro são gerados. Ao ler um registro de sua partição local, uma instância do filtro *Reader* irá atribuir um identificador interno ao processo por meio da fórmula $id = totalDeInstancias * sequencia + rank$.

Assumindo que as mensagens originadas por uma mesma instância do filtro *Reader* sempre serão recebidas em ordem pelo filtro *Blocking*, podemos dizer que este filtro sempre receberá registros com identificadores internos crescentes. Em um ambiente onde os filtros *Reader* estão bem balanceados, a diferença entre a ordem dos identificadores de registros que chegam até o filtro *Blocking* será pequena. Assumindo que poderá ocorrer casos onde essa ordem não será respeitada, mas que quase sempre ela é, podemos dizer que cada registro recém-chegado ao filtro *Blocking* terá um identificador interno maior do que todos os outros da lista invertida com a mesma chave de blocagem na maioria das vezes.

Com isto, a escolha feita no filtro *Scheduler* se resume a enviar o par para a instância que gerou o registro com maior identificador interno. Note que essa decisão leva em conta que uma vez recebido o par, a instância do filtro *Comparador* irá identificar

que ela não tem ambos os registros e irá enviar seu próprio registro para a instância que tem o registro complementar.

6.4 Evidência da Localidade Espacial

A existência da localidade espacial pode ser determinada comparando-se o número total de seqüências únicas em um *trace* real e em um novo *trace* gerado a partir de uma permutação aleatória sobre o *trace* original [Almeida et al., 1996].

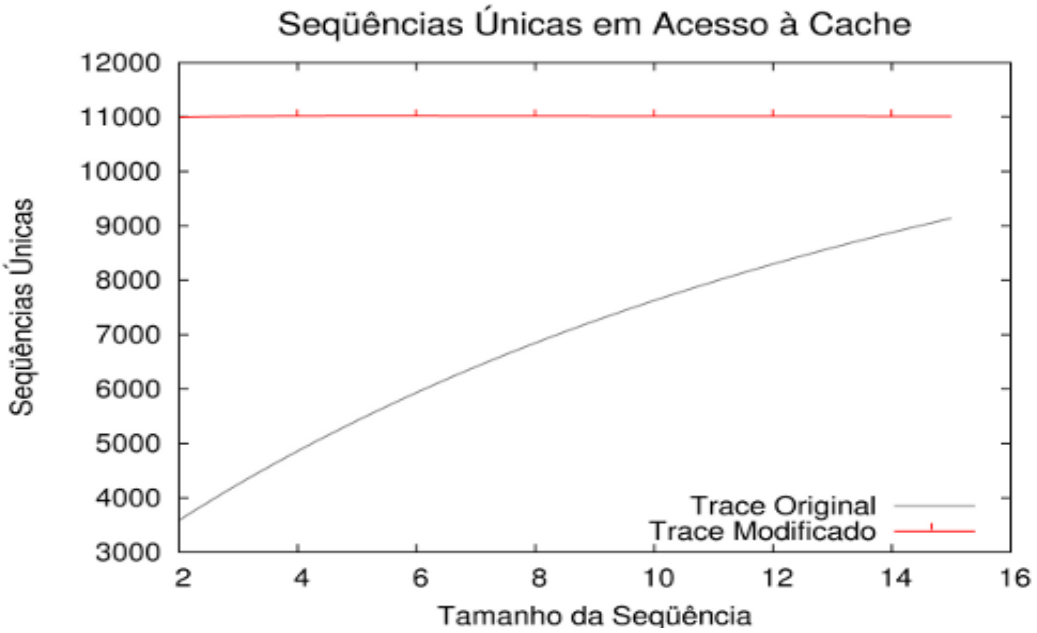


Figura 6.3. Seqüências únicas nos *traces* original e modificado

Para a geração do *trace*, consideramos o acesso aos registros da partição local de uma instância do filtro *Comparator*. A análise desses acessos permite que seja definido o tamanho dos *blocos da cache*. Sempre que um recurso é referenciado, toda a linha que contém esse recurso é trazida para a cache, amortizando o custo do acesso Snir & Yu [2005].

A Figura 6.3 mostra o número total de seqüências únicas em função do tamanho da seqüência k. Como se esperava, temos menos seqüências únicas no *trace* original do que no aleatoriamente permutado, indicando a presença da localidade espacial.

Seqüências de acesso seguem o mesmo padrão da geração dos pares na blocagem, como era de se esperar. Acreditamos que a definição da *blocos da cache* possa ser em função dos blocos onde cada um dos registros seria inserido. Neste momento, deixamos

a análise mais detalhada e a implementação de uma versão que explore a localidade de referência espacial para um trabalho futuro.

6.5 Sumário

Este capítulo apresentou as evidências sobre a presença da localidade de referência no algoritmo de pareamento de registros. A ênfase foi a localidade temporal. Discutimos como podemos explorar a localidade temporal na diminuição da comunicação de registros.

No próximo capítulo, apresentaremos os experimentos para avaliação da localidade de referência.

Capítulo 7

Avaliando a Localidade de Referência

Para a avaliação da nova versão da implementação que explora localidade de referência, usamos uma base de dados real de registros de internações hospitalares do município de Belo Horizonte. Ao todo, foram utilizados 600 mil registros. Desta vez, utilizamos uma blocagem mais restritiva, de forma a simular uma execução real. Foram gerados cerca de 90 milhões de pares candidatos (cerca de 150 comparações por registro), o que seria suficiente para identificar a maioria dos pares sem gerar um excesso de pares desnecessários. Os parâmetros de blocagem, funções e pesos de comparação se encontram no Apêndice B.

Os experimentos foram executados em um cluster com 14 máquinas Intel(R) Core(TM)2 CPU 2.13GHz, com 2Gb de memória principal, rede Gigabit Ethernet, e sistema operacional Linux 2.6. Todos os discos rígidos são SATA 7200 RPM e taxa de transferência (medida) aproximadamente de 90 Mb/s. Tomamos o cuidado para executar apenas uma instância do filtro *Comparator* em cada máquina.

7.1 Avaliando a Escolha da Instância para Comparação

Este experimento avalia como a escolha da instância influencia a execução do algoritmo. Pretendemos mostrar que a escolha correta da instância do filtro *Comparator* que receberá o par candidato para a comparação influencia o desempenho do algoritmo. Utilizamos duas estratégicas para a escolha da instância. Note que a estratégia 2 é a estratégia discutida na seção 6.3.3. Em ambos os casos, variamos o tamanho da *cache* entre 10 e 100%:

1. Enviar para a instância que possui o registro com o menor identificador, isto é, o registro que menos recentemente foi lido.

2. Enviar para a instância que possui o registro com o maior identificador, isto é, o registro mais recentemente lido.

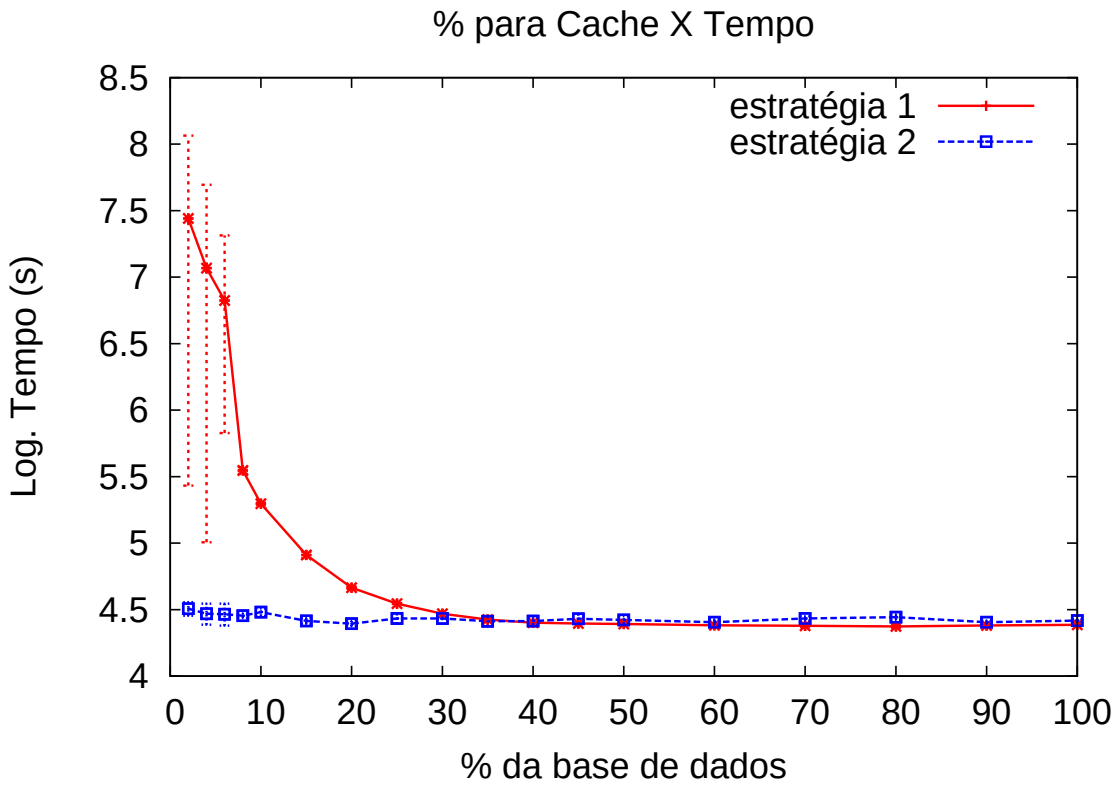
Esperamos que ao enviar o registro para a instância que possui o registro mais recentemente lido, um número maior de registros sejam comunicados e com isto o tempo de processamento também aumente. Ao enviar o par para a instância que possui o registro mais recentemente lido, estamos explorando a forma como os pares são gerados na implementação da blocagem. As Figuras 7.1 e 7.2 mostram os resultados do experimento. Podemos notar que as curvas geralmente se encontram ou ficam bem próximas quando o tamanho da cache é em torno de 20 a 30% do tamanho da base de dados. Este comportamento se repetiu para diferentes configurações do *cluster*. Concluímos que é possível manter o desempenho do algoritmo utilizando uma pequena parte dos dados armazenados *cache* para esse experimento, que, cabe lembrar, procurou ser fiel ao procedimento normal.

Para tamanhos da *cache* entre 2% e 30%, os tempos de execução para a primeira estratégia sempre foram superiores aos tempos da segunda, considerando-se um intervalo de confiança de 95%. A fim de entender as razões, considere os gráficos das Figuras 7.3, 7.4 e 7.5. Novamente, o comportamento quando usamos a primeira ou a segunda estratégia não varia muito quando mudamos a configuração do *cluster*. A quantidade de registros enviados entre instâncias do filtro *Comparator* para a segunda estratégia (utilizando maior identificador de registro) sofreu uma pequena oscilação quando variávamos o tamanho da *cache* se comparada à variação apresentada pela primeira estratégia.

Diferentemente das curvas dos gráficos de tempo, as curvas dos gráficos de quantidade de registros enviados para as duas estratégias se aproximam quando o tamanho percentual da *cache* está entre 40 e 60%, quando a eficiência da *cache* atinge o máximo. Supomos que o tamanho do registro utilizado (150 bytes) não foi suficiente para fazer com que a latência da rede fosse significativa a ponto dos gráficos de tempo e de registros enviados pudessem ser mais próximos, apesar de seguirem um mesmo padrão.

Praticamente, com o tamanho da *cache* maior do que 50% da base, não há mais variações na quantidade de registros comunicados, tendo portanto a cache atingido sua máxima eficiência. Após esse tamanho, a variação entre as duas estratégias é próxima de 0,5% em favor da segunda.

Figura 7.1. Tempo de execução para estratégias utilizando-se o registro menos recentemente lido (1) e registro mais recentemente lido (2) para a decisão do encaminhamento da mensagem, variando-se tamanho da *cache* (com 4 instâncias do do filtro *Reader*).

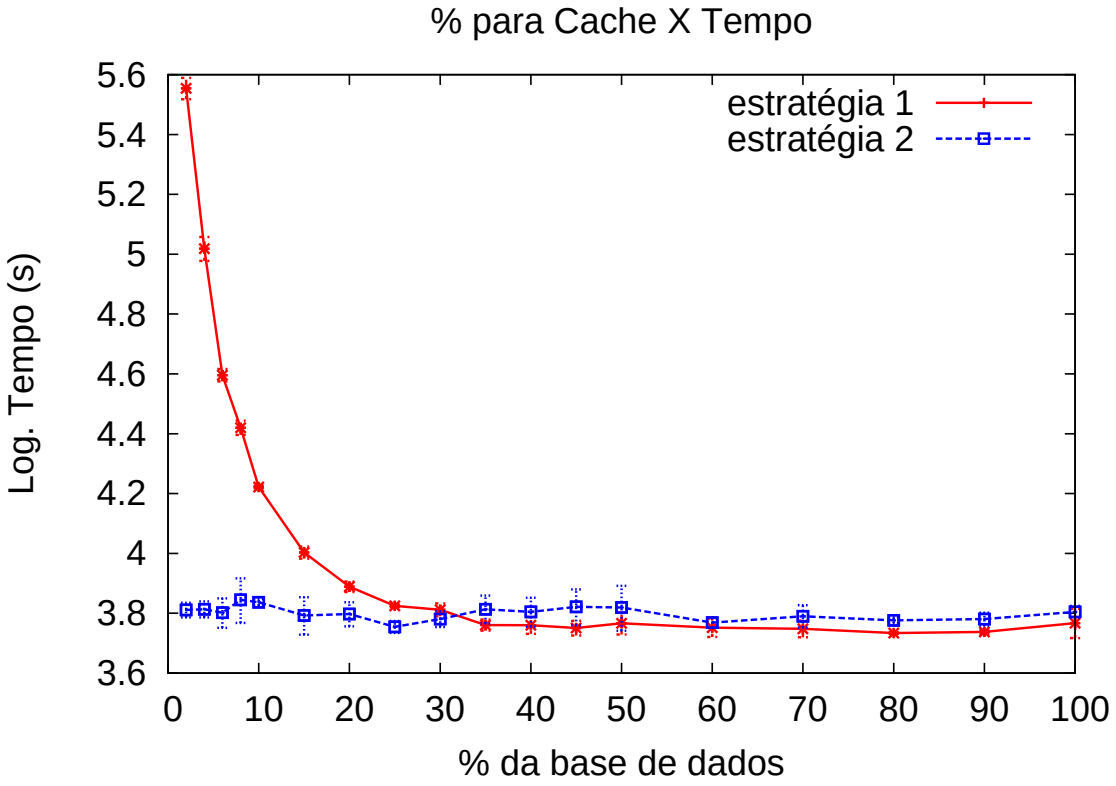


7.2 Utilizando a heurística baseada em grafos

Como visto no experimento anterior, a quantidade de registros trocados entre as instâncias do filtro *Comparator* influenciou o tempo de execução. Este experimento tenta explorar a heurística e avaliar se ela consegue diminuir o número de registros enviados e assim diminuir o tempo de execução. Como dissemos (seção 6.3.2.1), a heurística efetivamente diminui a quantidade de registros, além do custo de aplicá-la a todos os pares candidatos gerados faz com ela cause uma sobrecarga extra. Ao executar em uma rede rápida, como foi o caso deste experimento, os ganhos provenientes da redução do número de registros comunicados não superam os custos da sobrecarga causada pela heurística. Não foi possível realizar testes, mas possivelmente a heurística se tornaria uma opção mais interessante se o tempo para processá-la não superasse a latência da rede.

Neste experimento, utilizamos 300 mil registros e configurações do *cluster* variando de 2 a 6 máquinas. Comparamos a heurística baseada em grafos e sua execução

Figura 7.2. Tempo de execução para estratégias utilizando-se o registro menos recentemente lido (1) e registro mais recentemente lido (2) para a decisão do encaminhamento da mensagem, variando-se tamanho da *cache* (com 8 instâncias do do filtro *Reader*).



local, baseada na utilização do registro com maior identificador (mais recentemente lido). O tamanho máximo do grafo era de 1000 arestas. Todos os outros parâmetros foram os mesmos do experimento anterior.

Os gráficos de tempo, total de registros enviados e o percentual de redução alcançado com a utilização da heurística são apresentados nas Figuras 7.6, 7.7 e 7.8. A heurística baseada em grafo consegue reduzir a quantidade de registros enviados em aproximadamente 36%. Apesar desta redução, como esperávamos, o custo de processamento da heurística prejudicou o tempo de execução, ficando quase duas vezes maior quando utilizamos 6 instâncias. Isto ocorre porque há um atraso no *pipeline* que faz com que os filtros *Comparator* fiquem ociosos.

Concluímos que a heurística não traz ganhos significativos. Mesmo com registros maiores, o que é plenamente possível, a latência da rede ainda pode ser menor que o tempo de processamento. A escolha baseada no maior identificador de registro é simples e eficiente. Os experimentos utilizaram a blocagem clássica, mas qualquer implementação de outra técnica de blocagem que gere os pares agrupando-os pelo maior

Figura 7.3. Total de registros enviados entre instâncias do filtro *Comparator* variando-se tamanho da *cache*, 2 e 4 instâncias do filtro *Reader*

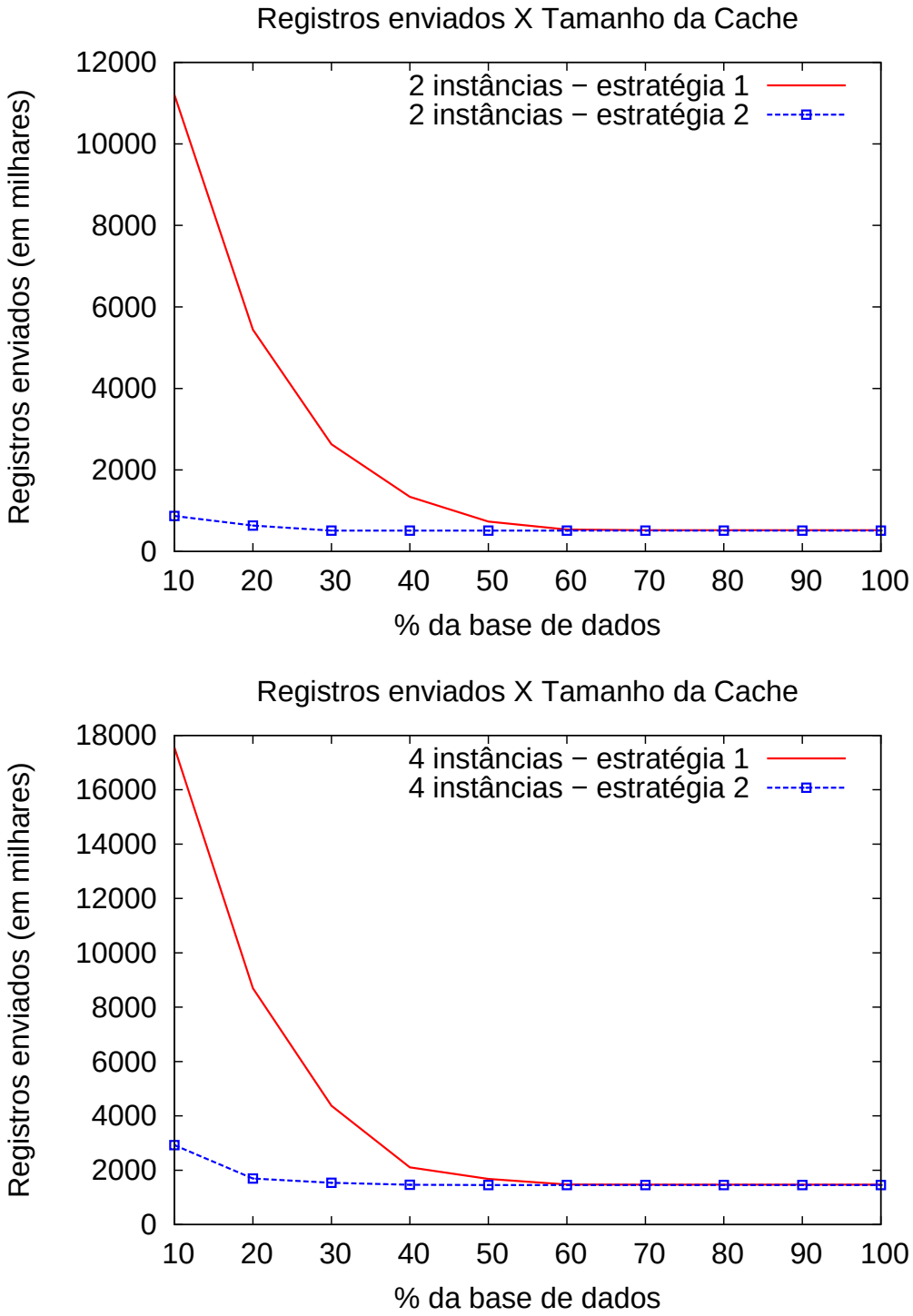


Figura 7.4. Total de registros enviados entre instâncias do filtro *Comparator* variando-se tamanho da *cache*, 6 e 8 instâncias do filtro *Reader*

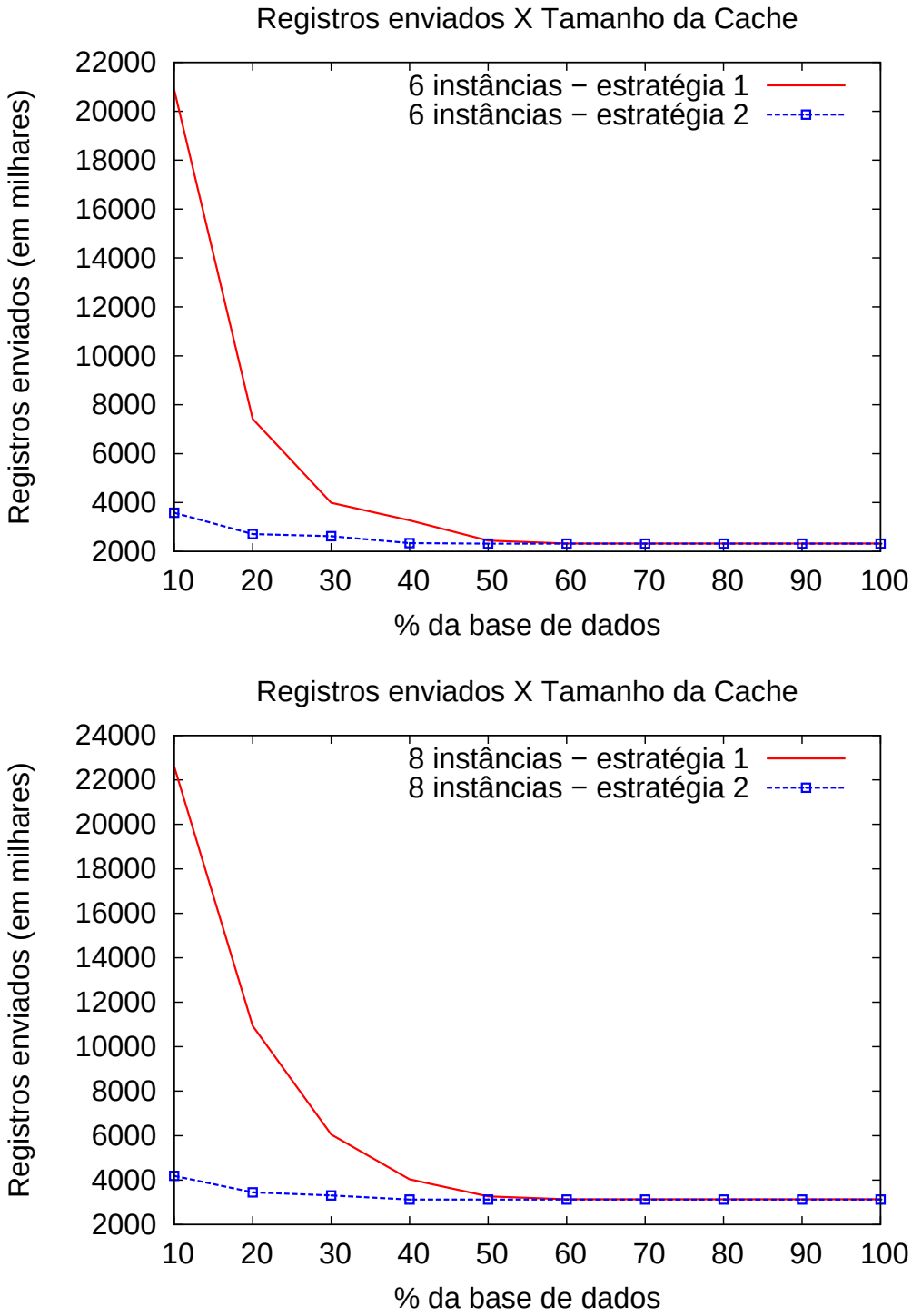
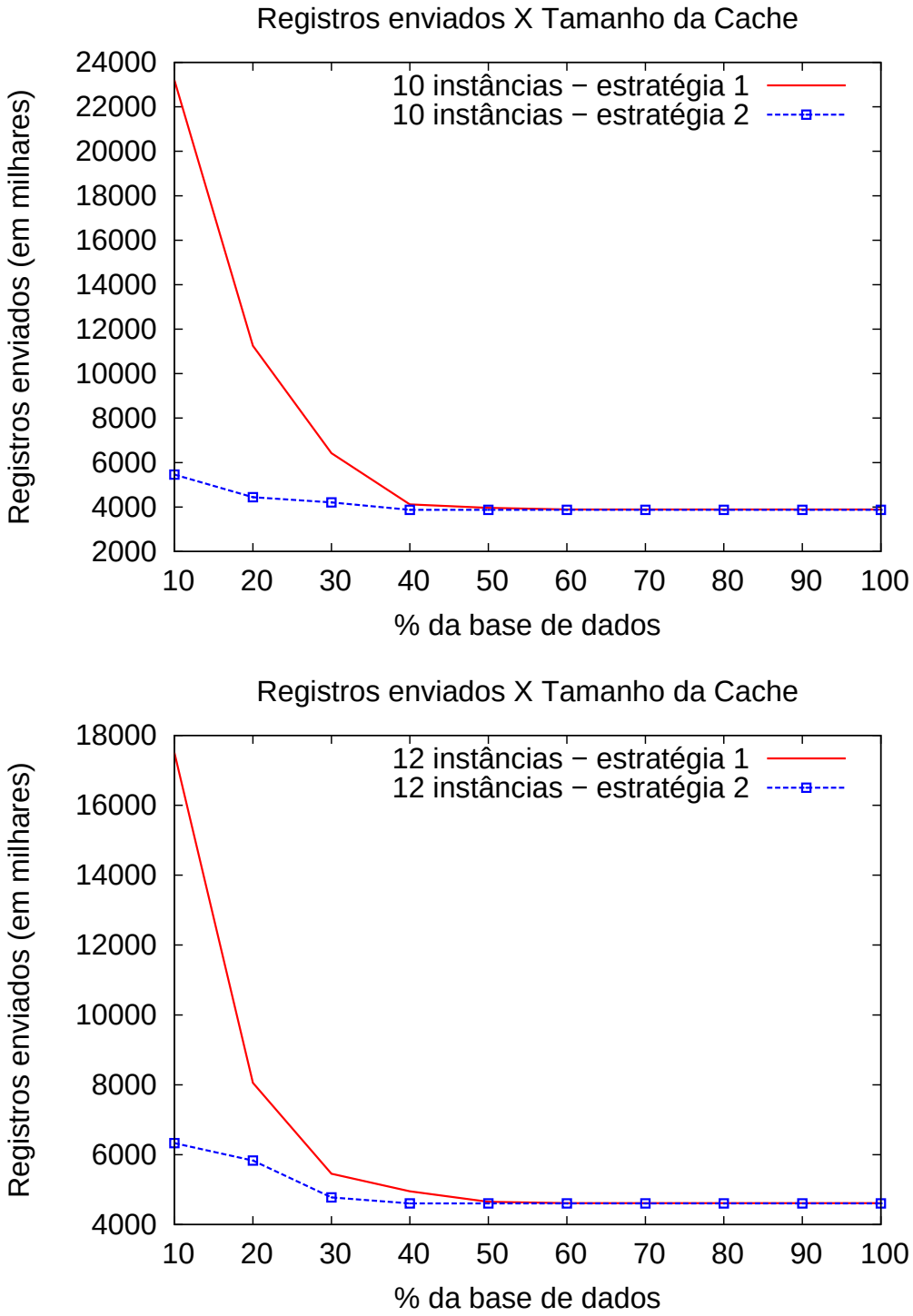


Figura 7.5. Total de registros enviados entre instâncias do filtro *Comparator* variando-se tamanho da *cache*, 10 e 12 instâncias do filtro *Reader*



identificador de registro poderá explorar a localidade de referência eficientemente.

7.3 Avaliando o Algoritmo com a Cache de Comunicação

A inclusão de uma política para a *cache* e a possibilidade de ter seu tamanho reduzido foram modificações feitas na primeira implementação, cujos testes foram apresentados no Capítulo 5. Este experimento pretende mostrar a escalabilidade do algoritmo quando usamos diferentes tamanhos da cache. Inicialmente, para avaliar a *cache de comunicação* executamos vários experimentos variando o seu tamanho entre 10% e 100% da base de dados.

Como percebemos nos gráficos das Figuras 7.9 e 7.10, o *speedup* da aplicação seguiu um comportamento similar, variando pouco quando variamos o tamanho da *cache*. Uma questão que pode surgir é se há necessidade de uma *cache*. Pelos experimentos, constatamos que uma cache de tamanho mínimo é necessária para a escalabilidade do algoritmo. A curva do tempo de execução da Figura 7.11 mostra que

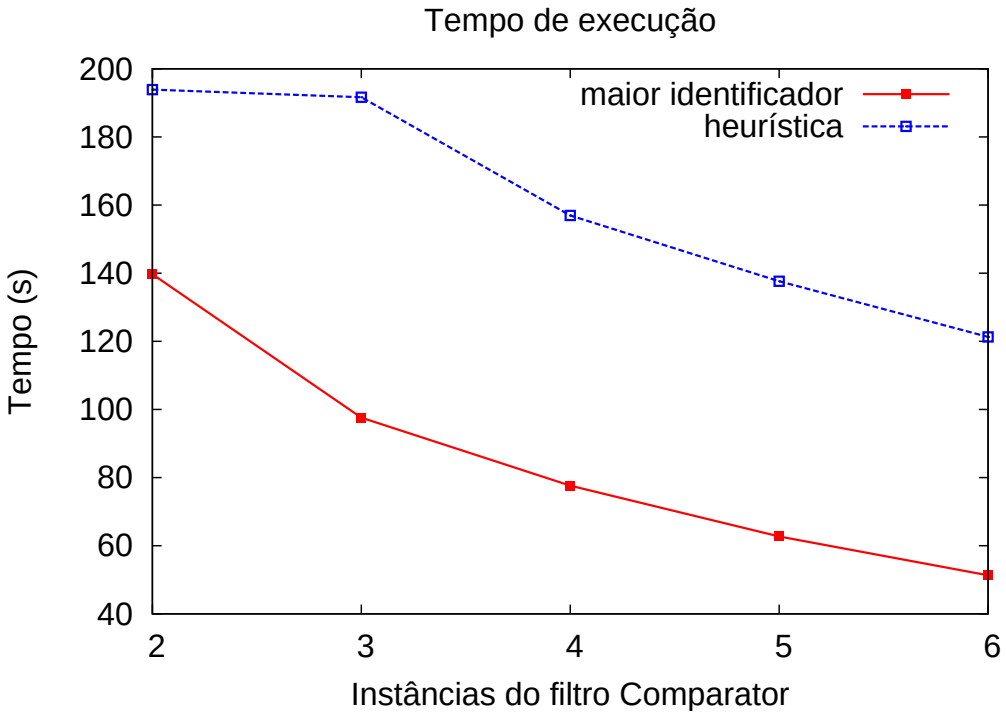


Figura 7.6. Comparação do tempo de execução do algoritmo quando utilizando-se a a heurística versus utilizando maior identificador de registro

Figura 7.7. Comparação do número de registros enviados durante a execução do algoritmo quando utilizando-se a a heurística versus utilizando maior identificador de registro

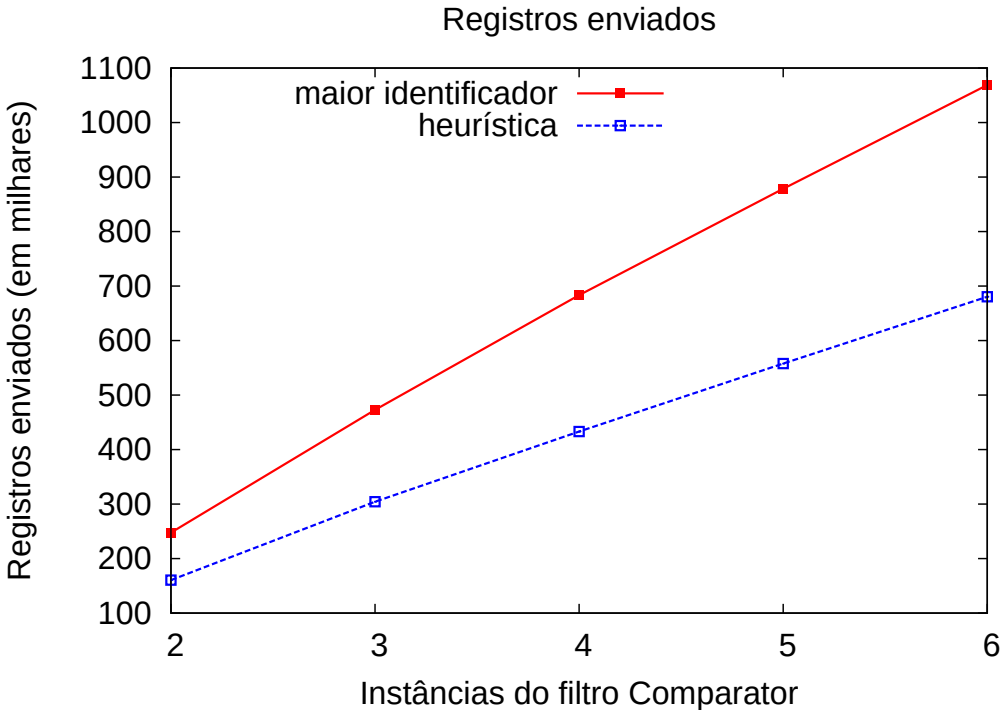


Figura 7.8. Comparação do percentual de redução no número de registros enviados utilizando-se a a heurística versus utilizando maior identificador de registro

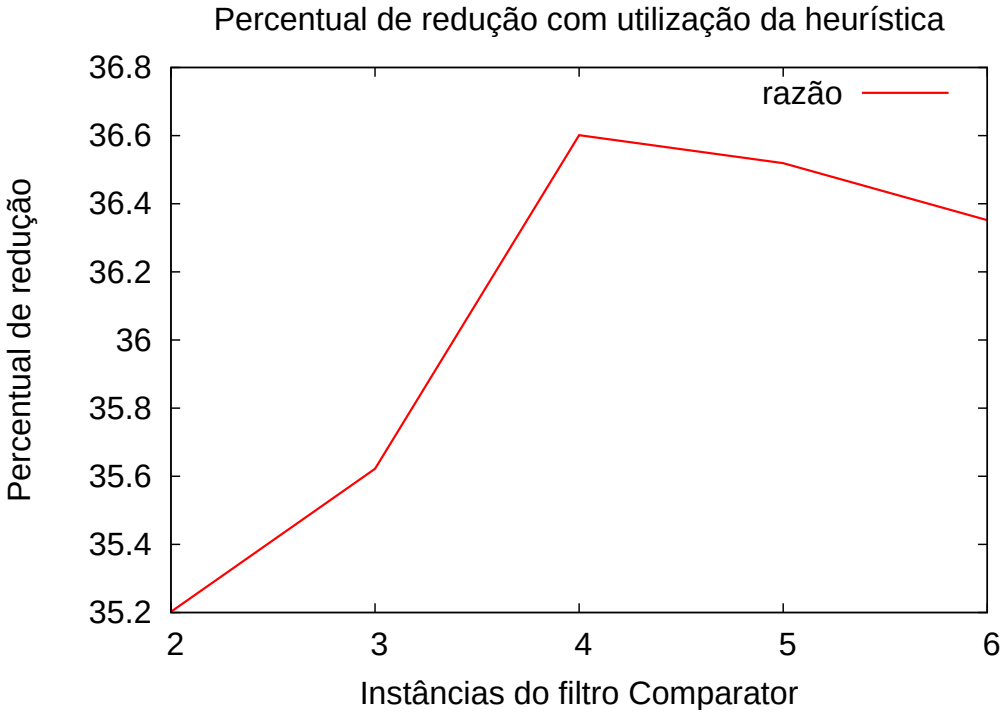


Figura 7.9. Speedup variando-se o tamanho da *cache*: 10% e 40% da base de dados

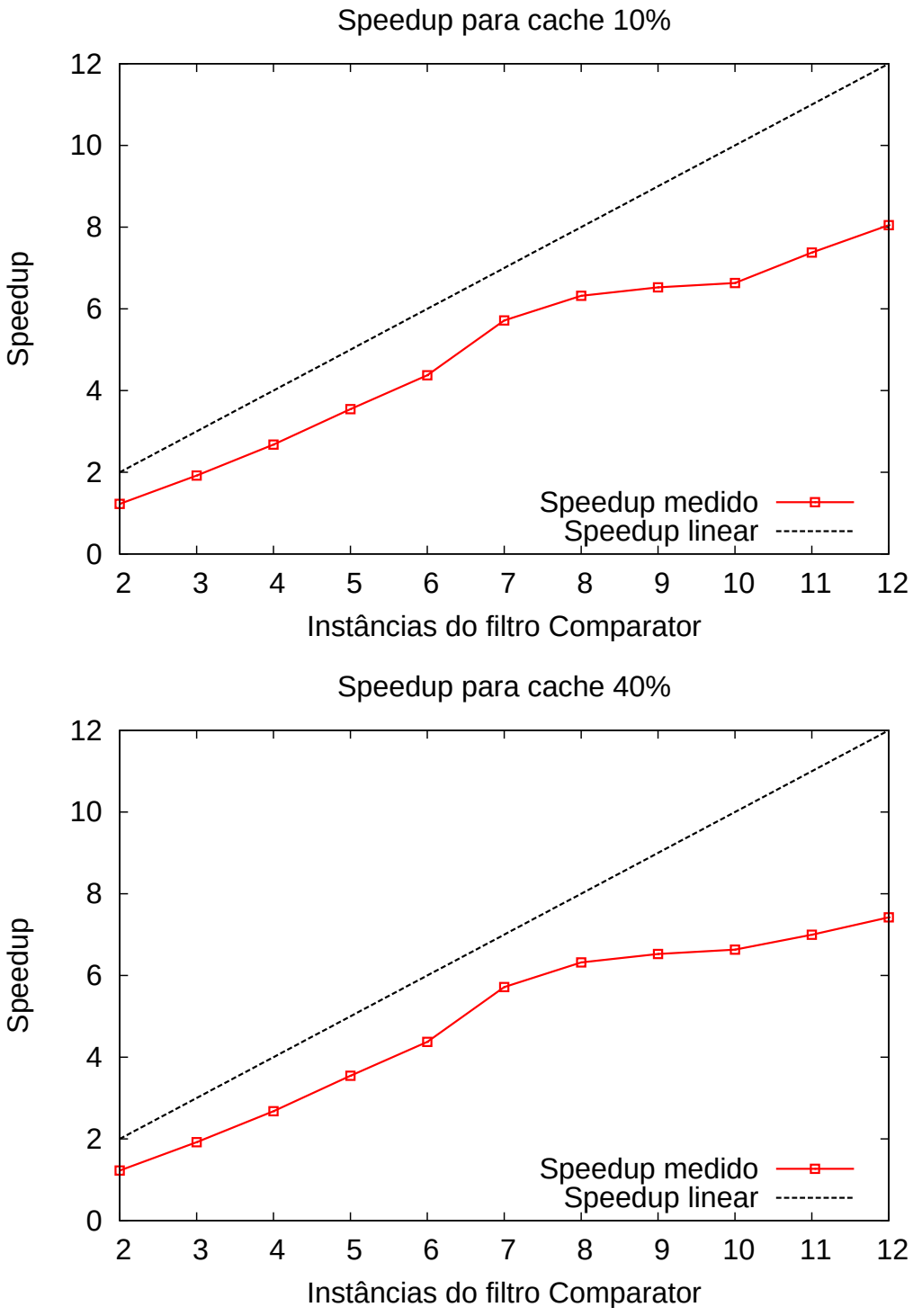
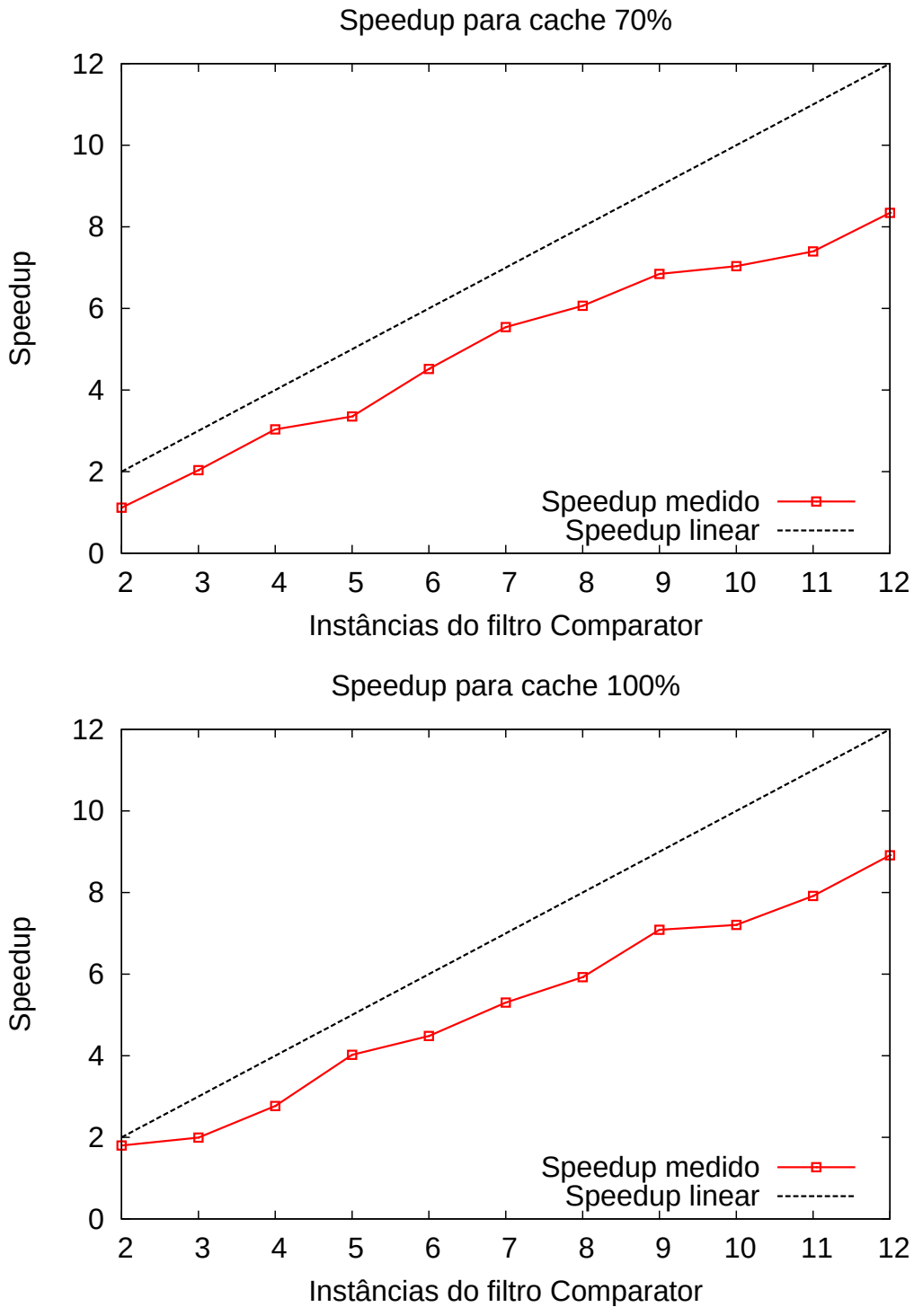


Figura 7.10. Speedup variando-se o tamanho da *cache*: 70% e 100% da base de dados



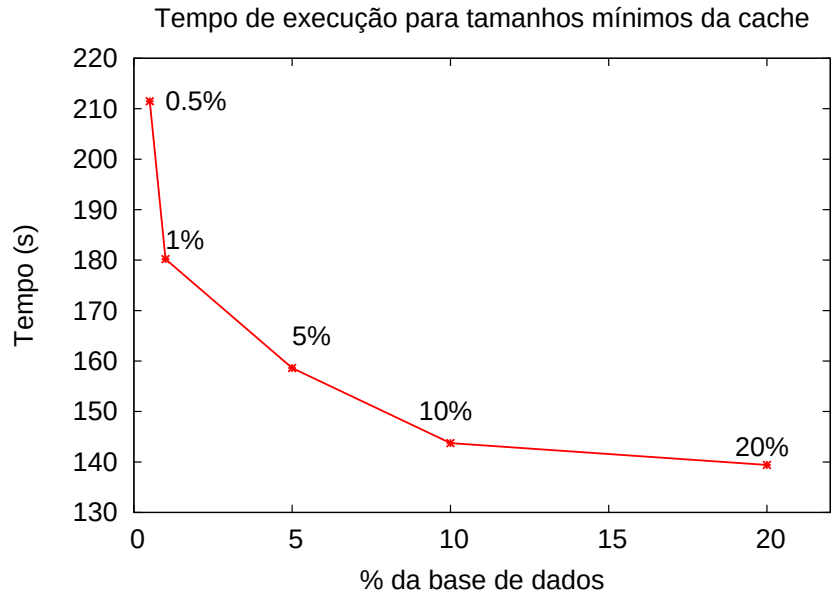


Figura 7.11. Tempo de execução para tamanhos mínimos da cache e 10 instâncias

o tempo cresce exponencialmente à medida que se diminui o tamanho percentual da cache. Todos os experimentos que foram executados sem a *cache* não terminaram. Eles falharam por sobrecarga do *pipeline*. O custo de comunicação fez com que os pares demorassem a ser processados e como a implementação buscou explorar o processamento assíncrono, os pares gerados pela blocagem foram acumulando-se até que os *buffers* de comunicação ficassem cheios.

7.4 Utilizando a Cache de Registros da Partição Local

Na implementação atual, a *cache* de registros da partição local armazena os registros lidos de um arquivo com registros de tamanho fixo. Para diminuir a influência de *buffers* do sistema operacional, utilizamos as funções `open()`, `lseek()` e `read()`.

Para avaliar a cache de registros da partição local, variamos o seu tamanho em relação ao tamanho total da partição de cada instância do filtro *Comparator* em 10, 50 e 100%. A *cache* de comunicação foi definida para 100%. Os testes foram realizados com 600 mil registros e cerca de 90 milhões de pares e com 4, 8 e 12 instâncias do filtro *Comparator*.

O resultado apresentado no gráfico da Figura 7.12 mostra que não houve variações significativas no tempo de execução quando reduzimos o tamanho da *cache* de registros da partição local de 100% para 10%. Mesmo para percentuais ainda menores

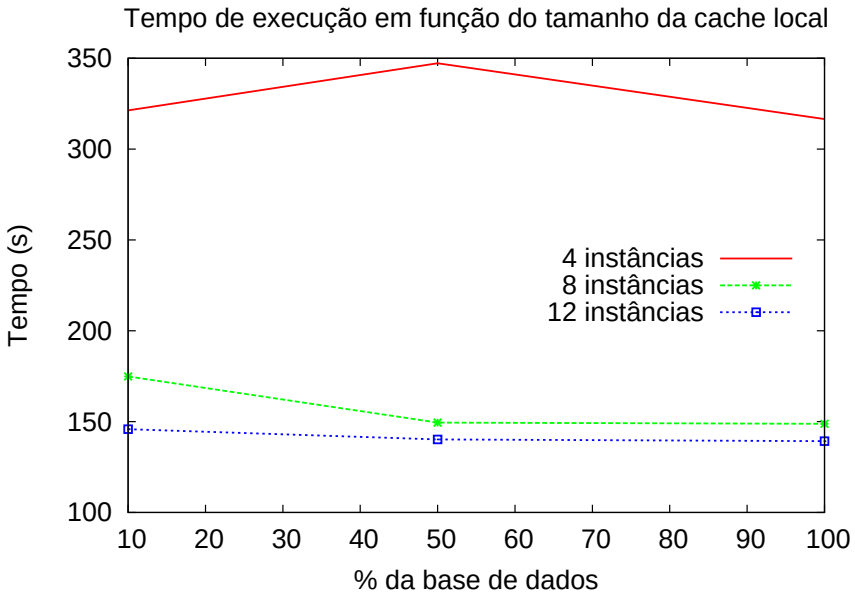


Figura 7.12. Tempo de execução em função do tamanho da *cache* partição de registros local

(1, 0,5 e 0,1%), o tempo praticamente foi o mesmo. Pela avaliação realizada, com a utilização de discos rápidos e partições de dados armazenadas no mesmo computador onde a instância do filtro *Comparator* é executada, podemos dizer que o ganho não é significativo quando se utiliza a *cache* de registros da partição local. Este resultado é importante, uma vez que a primeira implementação mantinha todos os dados em memória para manter a escalabilidade. Entretanto, uma *cache* mínima na aplicação se faz necessária, conforme visto nas Figuras 7.1 e 7.2. Todos os testes realizados sem a *cache* de registros da partição local falharam por sobrecarga do *pipeline* quando as instâncias passaram a atrasar o processamento em função da latência do disco.

Não foi possível executar testes utilizando outra forma de armazenamento dos dados (por exemplo, em um servidor de banco de dados). Acreditamos que a *cache* de registros da partição local poderá ser necessária quando o acesso aos dados tiver uma latência maior, como seria o caso de comunicação pela rede ou mesmo em um disco lento.

7.5 Sumário

Neste capítulo avaliamos a utilização de uma *cache* e como a escolha da instância que receberá o par candidato influencia o resultado do algoritmo. Ao utilizarmos a escolha pelo menor identificador, o agrupamento de pares com um registro comum gerado na

blocagem é desfeito. Com isto, a quantidade de registros a serem comunicados aumenta, bem como o tempo de execução.

Ao utilizar o maior identificador, ou seja, considerar o agrupamento de pares da blocagem, obtivemos resultados muito próximos quando variávamos o tamanho da *cache*. Isto possibilitou a utilização de uma *cache* bem pequena (10% do tamanho da base).

A heurística reduziu a quantidade de registros comunicados em cerca de 36% nos experimentos. Entretanto, o tempo de processamento superou a latência da rede e por isto a heurística não se mostrou uma boa escolha. Possivelmente, se a heurística puder ser aplicada somente aos pares candidatos que necessariamente demandem comunicação de registros ou a latência da rede for maior, haverá um ganho no tempo de processamento.

A *cache* de registros da partição local influenciou pouco o tempo de execução. Porém, ela é indispensável. Ao definirmos uma *cache* de registros da partição local com tamanho zero, a latência do disco fez com que ocorresse uma contenção nos filtros e com isto a execução abortava. Acreditamos que para um cenário onde o tempo de acesso aos dados seja significativo (por exemplo, com os dados armazenados em um servidor de gerenciamento de banco de dados), a utilização da *cache* de registros da partição local poderá ser fundamental.

Capítulo 8

Conclusão

Este trabalho apresentou um algoritmo paralelo eficiente para o problema de pareamento de registros (Capítulo 4), descrevendo cada uma das etapas do processo e como foi realizada a paralelização. Utilizou-se a teoria clássica de pareamento de registros [Fellegi & Sunter, 1969] e os resultados preliminares do pareamento foram avaliados para bases de dados sintéticas, onde os pares verdadeiros já eram conhecidos.

A implementação do algoritmo gerou a ferramenta FERAPARDA. Esta ferramenta é modular, permitindo que novas etapas do processo sejam adicionadas ou que as existentes sejam substituídas por novas implementações. A ferramenta FERAPARDA possui diferentes funções de comparação de caracteres, possibilitando a comparação de registros com presença de erros de entrada.

Os experimentos mostraram que a implementação está apta a processar muito mais pares em menos tempo do que os trabalhos relacionados. Foram utilizadas bases de dados sintéticas e reais, de diferentes tamanhos e, em ambos os casos, obtivemos uma boa escalabilidade. Foi possível construir a aplicação de forma que ela seja independente da disposição inicial dos dados. Consideramos que isto é bom porque o processo normal de pareamento envolve várias execuções de ajuste nos parâmetros. Ter que reorganizar os dados a cada novo experimento poderia ser contraproducente. Apesar de não terem sido feitos testes que comprovassem, acreditamos que é possível organizar os dados para se obter melhores resultados e a aplicação já está pronta para isto.

A comunicação entre filtros é intensa e justificou o estudo sobre localidade de referência e utilização de uma *cache* (Capítulo 6). Os experimentos realizados explorando a localidade de referência mostraram que é possível processar grandes bases de dados usando pouca memória física a cada momento (Capítulo 7), sem perda significativa de desempenho. O *speedup* com o uso de *caches* menores é bom, variando sua eficiência entre 60 a 75% para um cluster de até 12 máquinas. Um resultado importante

é que a seqüência de geração dos pares candidatos na blocagem influencia diretamente a localidade de referência. A geração de pares deve ter como um dos objetivos agrupar o máximo de pares formados por um registro comum, a fim de explorar a localidade de referência. A heurística apresentada (Capítulo 6) realmente diminuiu a quantidade de registros comunicados, mas como o percentual de comparações onde há comunicação de registros é muito pequeno e a latência da rede é menor que o tempo de processamento da heurística, aplicá-la a todos os pares gerou uma sobrecarga desnecessária para o cenário dos experimentos.

A utilização do ambiente de execução Anthill nos permitiu explorar diferentes aspectos do paralelismo, como paralelismo de tarefas, de dados e assincronia. A assincronia em especial é um ponto de melhoria. Um atraso em qualquer estágio do *pipeline* faz com que a aplicação acumule mensagens até que os *buffers* de comunicação estourem. A nova versão do Anthill, atualmente em desenvolvimento, permitirá otimizar a utilização da rede e irá eliminar o problema, introduzindo uma sincronização transparente para a aplicação.

8.1 Trabalhos Futuros

Novas oportunidades de pesquisa e implementadas novas funcionalidades poderão ser incluídas na ferramenta, permitindo melhorar os resultados do processo de pareamento de registros. Entre as principais melhorias, destacamos:

1. **Implementar outras técnicas de blocagem:** Os trabalhos relacionados às técnicas de blocagem geralmente tratam aspectos sobre a cobertura dos pares verdadeiros, total geral de pares candidatos e os passos de construção das chaves e recuperação dos registros (ver Capítulo 3). Não existe nenhuma discussão sobre a paralelização da etapa de blocagem. Um estudo sobre como paralelizar cada técnica de blocagem e como a recuperação de registros impacta no fluxo do *pipeline* pode incluir uma nova dimensão na avaliação das técnicas. Aparentemente, técnicas como *sorting blocking* [Hernandez & Stolfo, 1998], que têm o passo de recuperação de registros iniciada somente após a geração de todas as chaves, não deve explorar a assincronicidade.

2. **Avaliar técnicas de aprendizagem para a blocagem e para a definição dos demais parâmetros do processo:** Trabalhos como de Carvalho et al. [2006] e Winkler & Winkler [2000] propõem a utilização de algoritmos que permitem a descoberta de valores ideais para os parâmetros e funções de comparação do processo de pareamento de registros. O primeiro utiliza a programação

genética para inclusive determinar quais seriam as funções de comparação de caractere ideiais e o segundo utiliza o algoritmo *EM* para determinar os pesos do modelo proposto por Fellegi e Sunter. Acreditamos que uma comparação entre essas técnicas diminuiria a necessidade de conhecimento profundo sobre os dados, diminuindo a importância de um especialista.

3. **Incluir uma etapa para aplicar a transitividade entre pares verdadeiros:** A transitividade entre pares verdadeiros pode melhorar o resultado final, especialmente quando os registros possuem poucos atributos discriminativos e/ou ausência de informação. Entretanto, para a inclusão dessa nova etapa ao *pipeline*, é importante analisar se a aplicação da propriedade de transitividade a todos os pares é algo que compensaria o tempo extra de processamento ou se seria melhor utilizar uma blocagem menos restritiva.

4. **Inclusão de novas abordagens de pareamento**: O pareamento clássico [Fellegi & Sunter, 1969] não é a única técnica que poderia ser aplicada para o pareamento de registros. Recentemente, técnicas de *clustering* [Bhattacharya et al., 2006; Bilenko, 2006], utilização de grafos [Chen et al., 2007] e variações no pareamento clássico, como o classificador flexível [Peter Christen, 2004] têm sido propostas. Especialmente a utilização de grafos permite modelar aspectos semânticos no pareamento. Um exemplo é a representação de autores, artigos, conferências, instituições e outras entidades como vértices e as relações entre eles como arestas.

5. **Melhorar a implementação para evitar a contenção em alguns filtros:** Como já foi dito, este é atualmente o principal problema da implementação. Qualquer elemento da configuração de um *cluster* que apresente algum tipo de problema e cause uma latência maior do que os demais causará um acúmulo de mensagens e estouro dos *buffers* de comunicação.

6. **Explorar a localidade espacial:** Neste trabalho apenas mostramos que há evidências da localidade de referência espacial. Entendê-la pode ser útil especialmente quando o tempo de acesso aos registros for significativo.

Diversas oportunidades surgem para a ferramenta FERAPARDA. Pretendemos incluir uma interface para administração e execução do pareamento, criar a documentação de usuário, adicionar o suporte a bancos de dados relacionais e criar um portal colaborativo para a ferramenta. Acreditamos que exista um grande potencial para a aplicação dos resultados desta dissertação, especialmente por parte de órgãos governamentais.

Referências Bibliográficas

Acheson, E. (1968). Record linkage in medicine. In *Proceedings of the international symposium*. Oxford University Press.

Almeida, V.; Bestavros, A.; Crovella, M. & de Oliveira, A. (1996). Characterizing reference locality in the WWW. In *DIS '96: Proceedings of the fourth international conference on on Parallel and distributed information systems*, pp. 92--107, Washington, DC, EUA. IEEE Computer Society.

Baxter, R.; Christen, P. & Churches, T. (2003). A comparison of fast blocking methods for record linkage. In *Proceedings of 9th ACM SIGKDD Workshop on Data Cleaning, Record Linkage and Object Consolidation*.

Benjelloun, O.; Garcia-Molina, H.; Menestrina, D.; Whang, Q. S. S. E.; Widom, J. & Jonas, J. (2005). Swoosh: A generic approach to entity resolution. Relatório Técnico, Stanford InfoLab.

Bhattacharya, I.; Licamele, L. & Getoor, L. (2006). Relational clustering for entity resolution queries. In *ICML 2006 Workshop on Statistical Relational Learning (SRL)*, Pittsburgh, PA, EUA.

Bilenko, M.; Kamath, B. & Mooney, R. J. (2006). Adaptive blocking: Learning to scale up record linkage. In *ICDM '06: Proceedings of the Sixth International Conference on Data Mining*, pp. 87--96, Washington, DC, EUA. IEEE Computer Society.

Bilenko, M. Y. (2006). Learnable similarity functions and their application to record linkage and clustering. Master's thesis, Faculty of the Graduate School of The University of Texas at Austin.

Chen, Z.; Kalashnikov, D. V. & Mehrotra, S. (2007). Adaptive graphical approach to entity resolution. In *JCDL '07: Proceedings of the 7th ACM/IEEE joint conference on Digital libraries*, pp. 204--213, Nova Iorque, EUA. ACM.

Cherchiglia, M. L.; Guerra Júnior, A. A.; Andrade, E. I. G.; Machado, C. J.; Acúrcio, F. d. A.; Meira Júnior, W.; Paula, B. D. & Queiroz, O. V. (2007). A construção da base de dados nacional em Terapia Renal Substitutiva (TRS) centrada no indivíduo: aplicação do método de linkage determinístico-probabilístico. *Revista Brasileira de Estudos de População*, 24:163 – 167.

Christen, P. (2005). Probabilistic data generation for deduplication and data linkage. In *IDEAL*, pp. 109–116.

Christen, P. & Goiser, K. (2007). *Quality and Complexity Measures for Data Linkage and Deduplication*, volume 43. Springer Berlin / Heidelberg, Secaucus, NJ, EUA.

Datasus (2008). Procedimentos hospitalares do SUS - por local de residência - Brasil. Disponível em http://tabnet.datasus.gov.br/cgi/tabcgi.exe?sih/cnv/pruf.def.

de Carvalho, M. G.; Gonçalves, M. A.; Laender, A. H. F. & da Silva, A. S. (2006). Learning to deduplicate. In *JCDL '06: Proceedings of the 6th ACM/IEEE-CS joint conference on Digital libraries*, pp. 41--50, Nova Iorque, EUA. ACM.

Denning, P. J. & Schwartz, S. C. (1971). Properties of the working-set model. In *SOSP '71: Proceedings of the third ACM Symposium on Operating Systems Principles*, p. 130, Nova Iorque, EUA. ACM.

Drumond, E. F. & Machado, C. J. (2008). Linkage entre registros do SIHSUS e SINASC: possíveis vieses decorrentes do não pareamento. *Revista Brasileira de Estudos da População*.

Fellegi, I. P. & Sunter, A. B. (1969). A theory for record linkage. *Journal of the American Statistical Association*, 64(328):1183--1210.

Ferreira, R. A.; Wagner Meira, J.; Guedes, D.; Drummond, L. M. A.; Coutinho, B.; Teodoro, G.; Tavares, T.; Araujo, R. & Ferreira, G. T. (2005). Anthill: A scalable run-time environment for data mining applications. In *SBAC-PAD '05: Proceedings of the 17th International Symposium on Computer Architecture on High Performance Computing*, pp. 159--167, Washington, DC, EUA. IEEE Computer Society.

Gill, L. (2001). Methods for automatic record matching and linkage and their use in national statistics. *National Statistics Methodological Series no. 25*. National Statistics Methodological Series No.25, Office for National Statistics. Inglaterra.

Gropp, W.; Lusk, E.; Doss, N. & Skjellum, A. (1996). A high-performance, portable implementation of the MPI message passing interface standard. *Parallel Computing*, 22(6):789--828.

Gu, L. & Baxter, R. A. (2004). Adaptive filtering for efficient record linkage. In *SDM*.

Hernandez, M. A. & Stolfo, S. J. (1998). Real-world data is dirty: Data cleansing and the merge/purge problem. *Data Min. Knowl. Discov.*, 2(1):9--37.

Jaro, M. A. (1989). Advances in record linkage methodology as applied to matching the 1985 census of tampa. In *Journal of the American Statistical Association*, volume 84(406), p. 414:420.

Kan, M.-Y. & Tan, Y. F. (2008). Record matching in digital library metadata. *Communications ACM*, 51(2):91–94.

Kawai, H.; Garcia-Molina, H.; Benjelloun, O.; Menestrina, D.; Whang, E. & Gong, H. (2006). P-swoosh: Parallel algorithm for generic entity resolution. Relatório Técnico, Stanford InfoLab.

Lee, D. & Kim, H. (2007). Parallel Linkage. Relatório Técnico, The Pennsylvania State University.

Mattson, R. L.; Gecsei, J.; Slutz, D. R. & Traiger, I. L. (1970). Evaluation techniques for storage hierarchies. *IBM Journal*, 9(2):78--117.

N. S. D'Andrea Du Bois, J. (1969). A solution to the problem of linking multivariate documents. In *Journal of the American Statistical Association*, volume 64, pp. 163–174. No. 325 (Mar., 1969).

Nathan, G. (1967). Outcome probabilities for a record matching process with complete invariant information. In *Journal of the American Statistical Association*, number 318 (Jun., 1967) in 32, pp. 454–469.

Newcombe, H. B. (1967). Record linking: The design of efficient systems for linking records into individual and family histories. *American Journal of Human Genetics*, 19(3):335–359.

Peter Christen, Tim Churches, M. H. (2004). Febrl: A parallel open source data linkage system. In *Springer Lectures Notes in Artificial Intelligence, Proceedings of the 8th Pacific-Asia Conference, PAKDD 2004*, volume 3056, pp. 638–647, Sidney, Austrália. Springer.

Santos, W.; Teixeira, T.; Machado, C.; Jr., W. M.; Ferreira, R.; Guedes, D. & Silva, A. S. D. (2007). A Scalable Parallel Deduplication Algorithm. *SBAC-PAD*, 0:79–86.

Serpro (2004). Registro de cadastros sociais ultrapassa população do país. Disponível em http://www.serpro.gov.br/noticias-antigas/noticias-2004/20040324_10.

Snir, M. & Yu, J. (2005). On the theory of spatial and temporal locality. Relatório Técnico, Technical Report DCS-R-2005-2564 - Computer Science Dept., Univ. of Illinois em Urbana-Champaign.

Spirn, J. (1976). Distance string models for program behavior. *Computer*, 9(11):14--20.

Sunderam, V. S. (1990). PVM: a framework for parallel distributed computing. *Concurrency, Practice and Experience*, 2(4):315--340.

Tepping, B. (1968). A model for optimum linkage of records. In *Journal of American Statistical Association*, volume 63 (324), pp. 1321–1332.

Winkler, W. E. (2006). Overview of record linkage and current research directions. Relatório Técnico, Statistical Research Division - U.S. Census Bureau.

Winkler, W. E. & Winkler, W. E. (2000). Using the em algorithm for weight computation in the fellegi-sunter model of record linkage. In *Proceedings of the Section on Survey Research Methods, American Statistical Association*, pp. 667--671.

Apêndice A

Atributos considerados pelo DsGen

Atributo	**D**escrição
rec_id	Identificador do registro, serve para avaliar se os pares verdadeiros foram encontrados. Um registro original tem identificador $rec-DD-org$, onde DD é um número seqüencial. Já as réplicas têm um identificador igual a $rec-XX-dup-Y$, onde XX é o identificador seqüencial do original e Y é o número da cópia.
given_name	Nome formado a partir de uma tabela de freqüência e com lista de variações
surname	Sobrenome formado a partir de uma tabela de freqüência
street_number	Endereço formado a partir de uma tabela de freqüência
address_1	Primeira parte do endereço, formado a partir de uma tabela de freqüência
address_2	Complemento do endereço
suburb	Bairro, formado a partir de uma tabela de freqüência e com lista de variações
postcode	Código postal formado a partir de uma tabela de freqüência
state	formado por Estados
date_of_birth	Data de nascimento, formada a partir de uma tabela de freqüência e variações na escrita
age	Idade, formada a partir de uma tabela de freqüência
phone_number	Telefone formado por códigos de área da Austrália e a partir de uma tabela de freqüência
soc_sec_id	Número do seguro social, gerado aleatoriamente
blocking_number	Informação sobre o bloco (ignorado)

Tabela A.1. Atributos dos registros gerados pelo DsGen

Apêndice B

Atributos, funções e parâmetros de comparação

Atributo	**F**unção de Comparação	**P**robabilidades
given_name	winkler	m=0,94 u=0,1
surname	winkler	m=0,94 u=0,1
address_1	jaro	m=0,9 u=0,25
address_1	jaro	m=0,9 u=0,25
suburb	jaro	m=0,9 u=0,25
zipcode	comp. exata	m=0,9 u=0,25
state	comp. exata	m=0,98 u=0,1
phone_number	comp. exata	m=0,95 u=0,05
date_of_birth	comp. exata	m=0,95 u=0,05

Tabela B.1. Parâmetros usados para comparação

Apêndice C

Configuração do pareamento usada para experimentos com localidade de referência

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<project name="linkage-aih-apac" task="deduplication">
    <data-sources>
<data-source id="aih" file="/scratch/walter/aih_35k.txt"
record-length="150">
            <fields>
                <field name="id" type="int" size="7"/>
                <field name="cnpj" type="string" size="14"/>
                <field name="cep" type="string" size="8"/>
                <field name="munic" type="string" size="6"/>
                <field name="nasc" type="string" size="8"/>
                <field name="sexo" type="string" size="1"/>
                <field name="nome" type="string" size="20"/>
                <field name="nomem" type="string" size="60"/>
                <field name="sobrenome" type="string" size="25"/>
                <field name="nw" type="string" size="1"/>
            </fields>
        </data-source>
    </data-sources>
    <!-- classic, sorting -->
    <blocking type="classic">
        <conjunction>
            <part field-name="sobrenome" size="4"
             start="0" transform="dmetaphone"/>
```

```
            <part field-name="nasc" size="4" start="0" />
        </conjunction>
<conjunction>
            <part field-name="nome" size="3" start="0"/>
            <part field-name="cep"/>
        </conjunction>
<conjunction>
            <part field-name="munic"/>
            <part field-name="sexo"/>
            <part field-name="nasc" start="4" size="2"/>
        </conjunction>
    </blocking>
    <classifier type="basic"
     apply-transitivity="simple" match-min-result="-15"
            not-match-max-result="0">
        <approx-string-comparator m="0.95" u="0.01347" missing="0"
            field1="nome" field2="nome" frequency-table="id"
            function="winkler"/>
        <approx-string-comparator m="0.95" u="0.03187" missing="0"
            field1="sobrenome" field2="sobrenome" frequency-table="id"
            function="winkler"/>
        <exact-string-comparator m="0.99" u="0.042311" missing="0"
            field1="cnpj" field2="cnpj"/>
        <exact-string-comparator m="0.95" u="0.00047" missing="0"
            field1="cep" field2="cep"/>
        <exact-string-comparator m="0.99" u="0.50372" missing="0"
            field1="sexo" field2="sexo"/>
        <exact-string-comparator m="0.95" u="0.00113" missing="0"
            field1="nasc" field2="nasc"/>
        <approx-string-comparator m="0.9" u="0.0222" missing="0"
            field1="nomem" field2="nomem" function="jaro"/>
    </classifier>
<cache type="record" size="35000" intra-readers="35000"/>
    <output histogram="histogram.dat" min="-10" max="100"
     format="pair" file="/var/tmp/result.tgz"/>
</project>
```