UNIVERSIDADE FEDERAL DE MINAS GERAIS
INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
ESPECIALIZAÇÃO EM INFORMÁTICA: ÁREA DE CONCENTRAÇÃO: GESTÃO DE TECNOLOGIA DA INFORMAÇÃO

EVISTON BORGES PINTO

DEDUPLICAÇÃO PARA REDUÇÃO DE DUPLICIDADE NA BASE DE DADOS DO CARTÃO NACIONAL DE SAÚDE DO SUS: UMA ANÁLISE DA VERSÃO 5 DO SISTEMA CADSUS.

Brasília
2019

EVISTON BORGES PINTO

DEDUPLICAÇÃO PARA REDUÇÃO DE DUPLICIDADE NA BASE DE DADOS DO CARTÃO NACIONAL DE SAÚDE DO SUS: UMA ANÁLISE DA VERSÃO 5 DO SISTEMA CADSUS.

Monografia apresentada ao Curso de Especialização em Informática do Departamento de Ciência da Computação do Instituto de Ciências Exatas da Universidade Federal de Minas Gerais, como requisito parcial para a obtenção do grau de Especialista em Informática.

Área de Concentração: Gestão de Tecnologia da Informação

Orientadora: Gisele Lobo Pappa

Brasília
2019

© Eviston Borges Pinto
Todos os direitos reservados

Ficha catalográfica elaborada pela Biblioteca do ICEx – UFMG

Pinto, Eviston Borges

P659d Deduplicaão para redução de duplicidade na base de dados do cartão nacional de saúde do SUS: uma análise da versão 5 do sistema CADSUSI / Eviston Borges Pinto – Brasília,2019.
x, 42 f., il.

Monografia (especialização) – Universidade Federal de Minas Gerais. Departamento de Ciência da Computação.
Orientadora: Giselle Lobo Pappa

1.Computação – Monografias. 2. Deduplicação. 3. Cartão Nacional de Saúde Brasileiro. 4. Sistema CADSUS._ 5. Master Patient Index. MPI. I. Orientadora. II. Título

CDU 519.6*

UNIVERSIDADE FEDERAL DE MINAS GERAIS

INSTITUTO DE CIÊNCIAS EXATAS
DEPARTAMENTO DE CIÊNCIA DA COMPUTAÇÃO
ESPECIALIZAÇÃO EM INFORMÁTICA: ÁREA DE CONCENTRAÇÃO GESTÃO EM TECNOLOGIAS DA INFORMAÇÃO

Uso de técnicas de deduplicação como forma de reduzir duplicidade de registros na base de dados do Cartão Nacional do SUS (CNS): Uma análise da situação atual do banco de dados da versão 5 do sistema CADSUS

EVISTON BORGES PINTO

Monografia apresentada aos Senhores:

Profa. Gisele Lobo Pappa
Orientadora
DCC - ICEx – UFMG

Prof. José Nagib Cotrim Árabe
DCC - ICEx - UFMG

Prof. José Marcos Silva Nogueira
DCC - ICEx - UFMG

Belo Horizonte, 15 de março de 2019

Dedico esse trabalho a Deus, minha família, mais precisamente à minha esposa Lana Gaia e meus filhos Tiago e Lucas, professores da Universidade Federal de Minas Gerais (UFMG), todo o corpo administrativo da Escola Nacional de Administração Pública (ENAP) e aos demais alunos deste curso de especialização.

Resumo

O Cartão Nacional de Saúde (CNS) refere-se a um banco com dados nacional com registros de indivíduos que tiveram contato com serviços ou ações de saúde oferta-dos pelo Sistema Único de Saúde (SUS) e dados dos profissionais que executam esses serviços e ações. Seu objetivo é identificar esses indivídulos de forma unívoca. Versões dos sistemas de informação que mantém essa base de dados vem sendo disponibiliza-das desde 1999, com estratégias e tecnologias distintas de solução. Isso acarretou um volume elevado de duplicidades de registros. Na escrita deste trabalho, a base de dados do Cartão Nacional de Saúde possuía cerca de 286 milhões de indivíduos regis-trados como vivos. Isso representa um volume superior aos 208 milhões da populacão brasileira estimada pelo Instituto Brasileiro de Geografia e Estatística (IBGE). Como uma base fundamental para a execução da estratégia de e-Saúde brasileira, é impor-tante tratar as duplicidades para aumentar a qualidade do monitoramento e permitir que as políticas públicas voltadas à saúde sejam formuladas com maior assertividade e eficiência. Esse trabalho apresenta uma contextualização do Sistema Único de Saúde, sobre o Cartão Nacional de Saúde e seu sistema de informação. Além disso, discute sobre o que é o processo de deduplicação e como ele é executado na base do CNS. O objetivo principal desse trabalho foi analisar a eficiência do processo de tratamento de duplicidades implantado na versão 5 do sistema CADSUS. Esse sistema, cuja versão citada foi publicada em 2014, é responsável por manter a base de dados do Cartão Nacional de Saúde. Para essa análise, foram executados dois processos de caracteri-zação (profiling) em momentos distintos, um em novembro de 2017 e outro em fevereiro de 2019, para avaliar a evolução da quantidade dos registros durante esse período e a qualidade das informações dos atributos utilizados no processo de deduplicação do Cartão.

Palavras-chave: Deduplicação. Cartão Nacional de Saúde Brasileiro. Sistema CADSUS. Sistema Único de Saúde. SUS. Master Patient Index. MPI

Abstract

The National Health Card (CNS) refers toto a national database with records of individuals who had contact with health services or actions offered by the Unified Health System (SUS) and data from the professionals who perform these services and actions. Its purpose is to univocally identify these individuals. Versions of the information systems that maintain this database have been made available since 1999, with distinct strategies and technologies of solution. This entailed a high volume of duplicate records. In the writing of this work, the National Health Card database had about 286 million individuals registered as alive. This represents a volume over 208 million of the Brazilian population estimated by the Brazilian Institute of Geography and Statistics (IBGE). As a fundamental basis for the implementation of the Brazilian e-Health strategy, it is important to treat duplicities to increase the quality of monitoring and to enable public health policies to be formulated with greater assertiveness and efficiency. This paper presents a contextualization of the Unified Health System, about the National Health Card and its information system. In addition, it discusses what the deduplication process is and how it runs on the basis of CNS. The main objective of this work was to analyze the efficiency of the duplication treatment process implemented in version 5 of the CADSUS system. This system, whose quoted version was published in 2014, is responsible for maintaining the National Health Card database. For this analysis, two profiling processes were performed at different times, one in November 2017 and another in February 2019, to evaluate the evolution of the quantity of the records during this period and the quality of the information of the attributes used in the process of deduplication of the Card.

Keywords: Deduplication. Brazilian National Health Card. CADSUS system. Unified Health System. SUS. Master Patient Index. MPI

Lista de ilustrações

Figura 1 – Arquitetura Estratégia e-Saúde no Brasil 15
Figura 2 – Municípios e Estados do Projeto Piloto do sistema Cartão SUS iniciado em 1999 (1) 16
Figura 3 – Processo de deduplicação de registros, adaptado de Christen(2) e Christen(3). 19
Figura 4 – Exemplo de Classificação com base nos intervalos definidos 23
Figura 5 – Arquitetura SOA - Cartão Nacional de Saúde. .24
Figura 6 – Representação do OHMPI em tempo de configuração e operação. . . 25
Figura 7 – Configuração de match do OHMPI no Cartão Nacional de Saúde . . . 26
Figura 8 – Rotinas específicas para comparação de atributos de pessoa no CNS 27
Figura 9 – Tela inicial do OHMPI implantado no Ministério da Saúde para tratar a base de dados do CNS. 28

Lista de tabelas

Tabela 1	–	Estrutura organizacional e decisória do SUS	13
Tabela 2	–	Exemplos de substituições e exclusões de caracteres ou palavras no processo de pré-processamento.	20
Tabela 3	–	Fontes de dados utilizadas na manutenção do MPI do CNS	26
Tabela 4		Completude dos campos dematch da entidade PESSOA nas caracterizações de 2017 e 2019	30
Tabela 5	–	Distribuição dos valores do atributo NOME.	31
Tabela 6	–	Distribuição dos valores do atributo NOME DA MAE.	32
Tabela 7	–	Distribuição dos valores do atributo SEXO	34
Tabela 8	–	Consolidação dos valores inválidos e nulos para os atributos que representam o Município de Nascimento da Pessoa no banco do CNS	35

Lista de abreviaturas e siglas

ABNT	Associação Brasileira de Normas Técnicas
ANS	Agência Nacional de Saúde
CIB	Comissão Intergestores Bipartite
CIT	Comissão Intergestores Tripartite
CNS	Cartão Nacional de Saúde
CONASEMS	Conselhos de Secretarias Municipais de Saúde
CONASS	Conselho Nacional de Secretário da Saúde
COSEMS	Conselhos de Secretarias Municipais de Saúde
CPF	Cadastro de Pessoa Física
GM/MS	Gabinete do Ministro, Ministério da Saúde
IBGE	Instituto Brasileiro de Geografia e Estatística
MPI	Master Patient Index
MS	MInistério da Saúde
OBS	Observação
OHMPI	Oracle Healthcare Master Person Index
SES	Secretaria Estadual de Saúde
SGOP	Sistema de Gestão de Operadores
SMS	Secretaria Municipal de Saúde
SOA	Service Oriented Architecture
SUS	Sistema Único de Saúde

Sumário

1 Introdução **10**

2 Revisão Bibliográfica **12**

2.1 Sistema Único de Saúde (SUS) **12**

2.2 Cartão Nacional de Saúde (CNS) **14**

2.2.1 Sistema CADSUS **16**

2.3 Processo de Deduplicação ou Pareamento de registros **17**

2.3.1 Pré-Processamento **19**

2.3.2 Indexação / Blocagem **21**

2.3.3 Comparação **22**

2.3.4 Classificação **22**

2.3.5 Avaliação **23**

3 O Processo de deduplicação no Cartão Nacional do SUS **24**

4 Resultado da análise e discussão **29**

4.1 Completude **29**

4.2 Acurácia / Precisão **30**

4.2.1 Nome **30**

4.2.2 Nome da Mãe **31**

4.2.3 Data de Nascimento **33**

4.2.4 Sexo **34**

4.2.5 Município de Nascimento **34**

4.2.6 CPF **35**

4.3 Discussão **35**

5 Conclusão **38**

Referências **40**

APÊNDICES **43**

Apêndice 1 .... **44**

1 Introdução

A saúde é fator essencial para o desenvolvimento pessoal, social e econômico de indivíduos. Além de ser fundamental para a qualidade de vida destes. Trata-se de um dever do Estado prover a promoção, prevenção e tratamento da saúde de todos (4). Tendo em vista que o Brasil é um país com proporções continentais, a execução de políticas públicas voltadas à saúde da população, independente de ser cidadão brasileiro ou estrangeiro, torna-se inviável sem o apoio de tecnologia da informação.

Um dos requisitos fundamentais para execução, monitoramento e melhoria das políticas de saúde é a identificação das pessoas beneficiadas por elas. A necessi-dade de identificação dos usuários dos serviços de saúde no Brasil possui registros datados da década de sessenta (1). No ano de 1996 o Cartão Nacional de Saúde (CNS), ou, simplesmente, Cartão do SUS, teve suas diretrizes estabelecidas. O CNS é uma base de dados federal, mantida por estados e municípios, cujo um dos principais objetivos é a identificação unívoca dos usuários do Sistema Único de Saúde (SUS) (Matta(5)). Essa identificação ocorre por meio da atribuição de um número, denominado Número do Cartão Nacional de Saúde, ou número do CNS.

O sistema de informação que mantém a base de dados do CNS é chamado CADSUS. Na escrita deste trabalho, o CADSUS encontrava-se em processo de consolidação, ou implantação, da versão 6, que contempla, além da estrutura da versão 5, a inclusão do módulo Sistema de Gestão de Operadores (SGOP). A versão 5 faz parte do escopo deste trabalho por se tratar de uma implementação com um modelo de deduplicação de dados, introduzido por um componente de Master Person Index (MPI) em sua arquitetura. As versões anteriores do CADSUS não possuíam mecanismos eficientes para tratamento de duplicidade. Isso resultou em milhões de registros de usuários do SUS, inseridos na base de dados do CNS, com um volume bem superior ao número da população nacional, apontados pelo censo do IBGE. Isso evidenciou um alto volume de duplicidades. Registros apontam um volume de 27% de registros duplicados na base de dados da versão piloto do sistema (6) (7).

Este trabalho propõe-se a descrever o Cartão Nacional de Saúde e seu sistema de informação CADSUS, com foco na versão 5, e discorrer sobre as ferramentas utilizadas, por esse sistema de informação, para execução de deduplicação dos registros de pessoas em sua base de dados. Por fim, é realizada uma análise da eficácia dessa solução de deduplicação através de uma caracterização, ou profiling, do banco de dados do CNS, mantida pelo sistema CADSUS. O objetivo é analisar, com uso de uma ferramenta de data quality[1], o volume estimado de registros duplicados na base

[1] A ferramenta utilizada foi a Informatica Analyst da empresa Informática® cujo Ministério da Saúde possui licença

de dados e comparar com as versões anteriores que não contemplavam uma solução tecnológica de deduplicação, implantada na versão 5 do CADSUS. A hipótese é de que o volume de duplicidades de registros, em torno de 27% da base, inseridos em versões anteriores a 5, tenha sido reduzido após a adoção de uma arquitetura com um componente de Master Person Index (MPI), adquirida pelo Ministério da Saúde. Essa arquitetura foi implantada em 2014, com o lançamento da versão 5 do CADSUS.

O trabalho está estruturado em uma seção de revisão bibliográfica, na qual são descritos o Sistema Único de Saúde, o Cartão Nacional de Saúde e o que é um processo de deduplicação de dados. O desenvolvimento é elaborado com uma descrição da estrutura atual do processo de deduplicação do CADSUS, implantada em sua versão 5. Uma discussão sobre a eficácia desse processo finaliza o desenvolvimento do texto. Dados do IBGE são utilizados para comparação da população nacional e o volume de registros da base de dados do CNS.

2 Revisão Bibliográfica

2.1 Sistema Único de Saúde (SUS)

Saúde é um direito assegurado na Constituição da República Federativa do Brasil de 1988. Seu artigo nº 196, sobrescrito abaixo, estabelece que:

> A saúde é direito de todos e dever do Estado, garantido mediante políticas sociais e econômicas que visem à redução do risco de doença e de outros agravos e ao acesso universal e igualitário às ações e serviços para sua promoção, proteção e recuperação. (8)

Além do artigo sobrescrito acima, a Constituição de 1988, em seu artigo n.º 196, cria o arcabouço para o Sistema Único de Saúde:

> Art. 198. As ações e serviços públicos de saúde integram uma rede regionalizada e hierarquizada e constituem um sistema único, organizado de acordo com as seguintes diretrizes:
>
> I - descentralização, com direção única em cada esfera de governo;
> II - atendimento integral, com prioridade para as atividades preventivas, sem prejuízo dos serviços assistenciais;
> III - participação da comunidade. (8)

Como uma das ações para assegurar o direito à Saúde e regulamentar o “sistema único”, especificado no artigo 198 da Constituição de 1988, foi promulgada Lei n.º 8.080 de 19 de setembro de 1990, com objetivo de regulamentar o Sistema Único de Saúde (SUS). Especificamente, essa lei regulamenta as ações do SUS em todo o território nacional com estabelecimento de diretrizes organizacionais e financeiras. Com isso, o SUS pode ser considerado como um conjunto de ações e serviços de saúde, prestados por órgãos e instituições públicas federais, estaduais e municipais, da Administração direta e indireta e das fundações mantidas pelo Poder Público. O SUS está pautado nos seguintes princípios doutrinários, segundo o portal do Ministério da Saúde (MINISTÉRIO DA SAÚDE(9)):

> **Universalização**: a saúde é um direito de cidadania de todas as pessoas e cabe ao Estado assegurar este direito, sendo que o acesso às ações e serviços deve ser garantido a todas as pessoas, independentemente de sexo, raça, ocupação ou outras características sociais ou pessoais.
>
> **Equidade**: o objetivo desse princípio é diminuir desigualdades. Apesar de todas as pessoas possuírem direito aos serviços, as pessoas não são iguais e, por isso, têm necessidades distintas. Em outras palavras, equidade significa tratar desigualmente os desiguais, investindo mais onde a carência é maior.
>
> **Integralidade**: este princípio considera as pessoas como um todo, atendendo a todas as suas necessidades. Para isso, é importante a integração de ações, incluindo a promoção da saúde, a prevenção de doenças, o tratamento e a reabilitação. Juntamente, o princípio de integralidade pressupõe a articulação

> da saúde com outras políticas públicas, para assegurar uma atuação intersetorial entre as diferentes áreas que tenham repercussão na saúde e qualidade de vida dos indivíduos.

A estrutura do SUS tem como diretriz organizacional a hierarquização. Com isso o Sistema é composto por Ministério da Saúde (MS), Secretarias Estaduais de Saúde (SES), Secretarias Municipais de Saúde (SMS), Conselhos de Saúde, Comissão Inter-gestores Tripartite (CIT), Comissão Intergestores Bipartite (CIB), Conselho Nacional de Secretário da Saúde (Conass), Conselho Nacional de Secretarias Municipais de Saúde (Conasems) e Conselhos de Secretarias Municipais de Saúde (Cosems). Essa estrutura principal evidencia a descentralização da execução das ações de promoção, proteção e recuperação da saúde.

Tabela 1 – Estrutura organizacional e decisória do SUS

Esfera	Gestor	Comissão Intergestores	Representação Gestores	Colegiado Participativo
Federal	Ministério da Saúde	Comissão Tripartite	CONASS	Conselho Nacional
			CONASEMS	
Estadual	Secretarias Estaduais	Comissão Bipartite	COSEMS	Conselho Estadual
Municipal	Secretarias Municipais			Conselho Municipal

Fonte: Adaptado com base na Lei 8.080/90

Por ser ter uma estrutura descentralizada, o SUS possui suas ações de pro-moção, proteção e recuperação da saúde que são executadas nas três esferas gover-namentais. Contudo, a maior concentração da execução das ações encontra-se nos municípios brasileiros. Isso representa que as ações do Sistema Único de Saúde estão presentes em 5570 municípios executores da política de saúde brasileira. Uma estru-tura composta por municípios com as mais diversas características e diversificações, principalmente, do ponto de vista demográfico e financeiro.

Os desafios para execução plena das ações de saúde são elevados. Apenas com uso de sistemas informatizados é possível integrar toda essa estrutura, de forma que seja possível planejar, executar e monitorar ações ou políticas públicas voltadas a esse assunto. Devido a isso, diversas ações são tomadas com intuito de informatização como um recurso estratégico para a execução desse dever do Estado que é a saúde da população.

2.2 Cartão Nacional de Saúde (CNS)

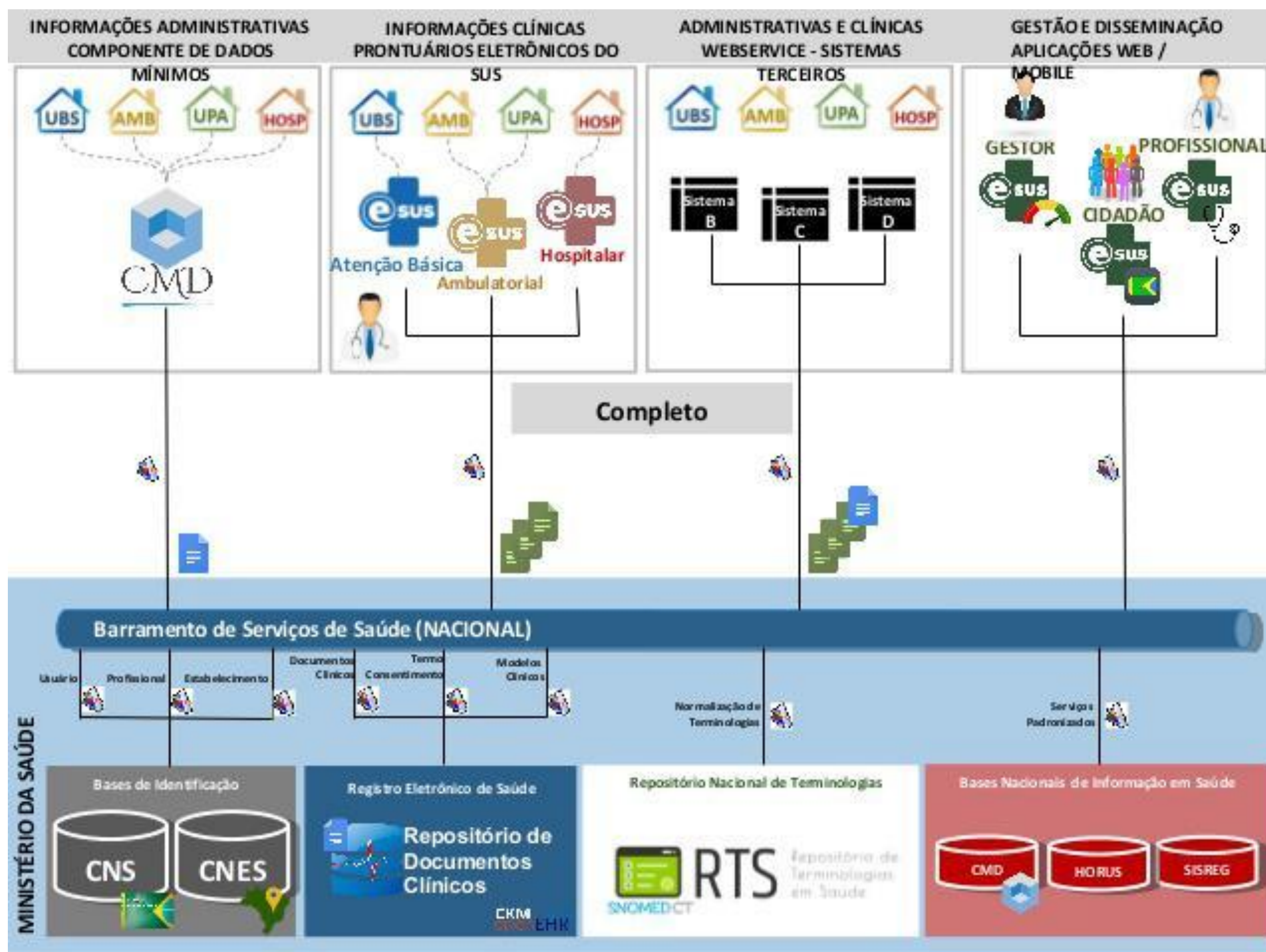
O Cartão Nacional de Saúde (CNS), ou Cartão SUS, regulamentado pela Porta-ria GM/MS n.º 940, de 28 de abril 2011, é um instrumento fundamental que possibilita que Estados, Municípios, Distrito Federal e União um melhor planejamento, execução, monitoramento de políticas de saúde no Brasil. Trata-se de uma base de dados responsável por identificar pessoas que tiveram contato com algum serviço do Sistema Único de Saúde (SUS) através de um número, assim como ocorre com o Cadastro de Pessoa Física (CPF). Seu uso não está restrito à identificação de usuários que utilizem serviços públicos voltados a promoção, proteção e recuperação da saúde. Por meio da Resolução Normativa 295/2012 da Agência Nacional de Saúde (ANS), que dispõe sobre a obrigatoriedade do número do Cartão Nacional de Saúde nos cadastros de beneficiários das operadoras de saúde suplementar, ou seja, planos de saúde, o número do CNS, também, passou a identificar os usuários da rede privada de saúde.

O CNS foi alicerçado por meio da Norma Operacional Básica do Sistema Único de Saúde, publicada no D.O.U. de 6/11/1996 (Saito et al.(10)). Por problemas observa-dos na implementação da NOB/SUS 01/96, após um abrangente processo de discussão entre os gestores, houve a publicação da Norma Operacional da Assistência Saúde 01/2001, instituída pela Portaria GM/MS n.º 95, de 26 de janeiro de 2001.

Contudo, as discussões sobre a necessidade de construir um sistema de forma a melhorar a aplicação das ações de saúde no Brasil, são mais antigas que essas resoluções. Há registros que datam da década de 60, em que o Ministro da Saúde, à época , declara sobre a "necessidade de haver no país um sistema de estatísticas que permitisse conhecer como cada cidadão brasileiro tinha atendidas suas necessidades de saúde, de forma organizada e individualizada." MINISTÉRIO DA SAÚDE(1).

O CNS compõe um dos alicerces da arquitetura da estratégia e-Saúde do Brasil (11), conforme Figura 1. A estratégia e-Saúde possui um comitê gestor definido por meio da Resolução n.º 5 da CIT, de 25 de agosto de 2016 (Brasil, Comissão Intergestores Tripartite(12)) e que é responsável, dentre outras competências, por definir a estratégia e-Saúde do Brasil. De forma sucinta, essa estratégia visa informatizar a estrutura de execução das ações de saúde do SUS, de forma integrada, com foco na interoperabilidade dos sistemas de informação que apoiam na execução dessas ações.

Figura 1 – CNS na estratégia e-Saúde do Brasil



8ª Assembléia do CONASS

Segundo Andrade et al.(13) :

> A implementação do Cartão Nacional de Saúde no âmbito do SUS já é uma realidade, **embora ainda existam problemas de duplicidade** e pouco uso efetivo. Para evitar duplicidade, esse cartão deveria ser associado a outros registros dos indivíduos. Existem algumas ações sendo tomadas desde 2011, relativas ao Ministério da Saúde, para melhorar a base de dados que estrutura o cartão – tais como sua integração com as bases de dados da Receita Federal, o sistema de óbitos da Previdência Social e os sistemas de nascidos vivos e de mortalidade do próprio MS –, sendo este projeto denominado interoperabilidade de bases de dados no âmbito do CNS. A deduplicação dos registros e as inativações dos registros indevidos e dos óbitos são ações que vêm sendo realizadas no âmbito federal e visam higienizar essa base de informações, tornando-a central na ligação entre usuários, profissionais de saúde e estabelecimentos, assim como de toda a assistência de saúde prestada aos cidadãos. Andrade et al.(13)

A duplicidade de registros é um relato constante na literatura encontrada sobre o Cartão Nacional de Saúde (13) (6)(7) e é um desafio para o corpo técnico e gestores do SUS que atuam na operação e construção dessa base de dados.

2.2.1 Sistema CADSUS

O sistema CADSUS representa o principal sistema de informação responsável por manter a base de dados do Cartão Nacional de Saúde. Foram encontrados registros de um projeto-piloto executado entre 1999 e 2002 (14) (6) que envolveu a implantação de um sistema em 44 municípios, contemplando cada uma das regiões do Brasil, conforme distribuição representada na .

Figura 2 – Municípios e Estados do Projeto Piloto do sistema Cartão SUS iniciado em 1999 (1)

Fonte: Retirado do Livro: Cartão SUS - O Brasil com Saúde

O resultado foi o registro de 13 milhões de usuários cadastrados na base do Cartão Nacional de Saúde. Além de algumas características técnicas, diversos problemas desse projeto-piloto são elencados na Nota Técnica CONASS n.º 22 de 14 de junho de 2011 (14). Essa nota técnica, também, discorre sobre o período após esse piloto, mais especificamente entre 2002 e 2009.

Após o projeto piloto supracitado, esforços para a implementação de outras versões do CADSUS (15) foram empreendidos. As versões iniciais tratavam as bases de identificação de forma decentralizada e offline, o que acarretou um significativo número de duplicidades devido a falta de sincronização “em tempo real” da base na-cional. Por exemplo, um cidadão em trânsito era cadastrado diversas vezes em cada unidade que o mesmo era atendido. A versão denominada CADSUS MULTIPLATA-FORMA, disponibilizada em 2006, considera-se a primeira versão do sistema em uma plataforma desenvolvida na linguagem JAVA. Outra versão foi denominada CADSUS STANDALONE. Ela já possuía uma integração com uma base nacional, porém, permitia operação offline, o que gerava duplicidades, conforme exemplificado acima.

Em 2010, uma proposta de revitalização do Cartão Nacional de Saúde foi apresentada em reunião da CIT. Nela, foram discutidos assuntos da solução e de marco regulatório. Com isso, foi decidido que a arquitetura para o novo sistema seria orientada a serviços, SOA[1]. Como um dos resultados desse projeto de revitalização, foi publicada a Portaria GM/MS n.º 940, de 28 de Abril de 2011, que regulamentou o Sistema Cartão Nacional de Saúde (Sistema Cartão). Isso proporcionou a base para versões que promoviam a interoperabilidade e uso de serviços.

A versão número 5 do CADSUS, publicada por volta de 2014, compreendeu uma ferramenta WEB[2] cuja base técnica foi composta por uma arquitetura orientada a serviço (SOA), como citado anteriormente, com utilização de solução de MPI (Master Patient Index) da empresa Oracle, denominada Oracle Healthcare Master Person Index, ou OHMPI (16). O objetivo do emprego dessa solução foi sanitizar a base de dados do CNS, tratando registros duplicados com uso de técnicas de deduplicação. A partir dessa versão, os usuários operadores têm à sua disposição um grupo de funcionalidades para identificação e unificação de registros de pessoas que tiveram contato com algum serviço do SUS.

No momento da escrita deste trabalho, a versão 6 do CADSUS foi implantada. Essa versão contempla a versão 5, com suas funcionalidades e arquitetura, conforme descrito acima, com a inclusão de um módulo denominado Sistema de Gestão de Operadores (SGOP) (17).

2.3 Processo de Deduplicação ou Pareamento de registros

O processo de deduplicação ou pareamento (Record Linkage) de registros consiste na tarefa de comparação desses com objetivo de classificá-los como duplicados, não duplicados ou possíveis duplicados com objetivo de reduzir o volume de

[1] Service-Oriented Architecture (Arquitetura Orientada a Serviço)
[2] World Wide Web (WWW, www ou Web) — sistema hipertextual que opera através da internet

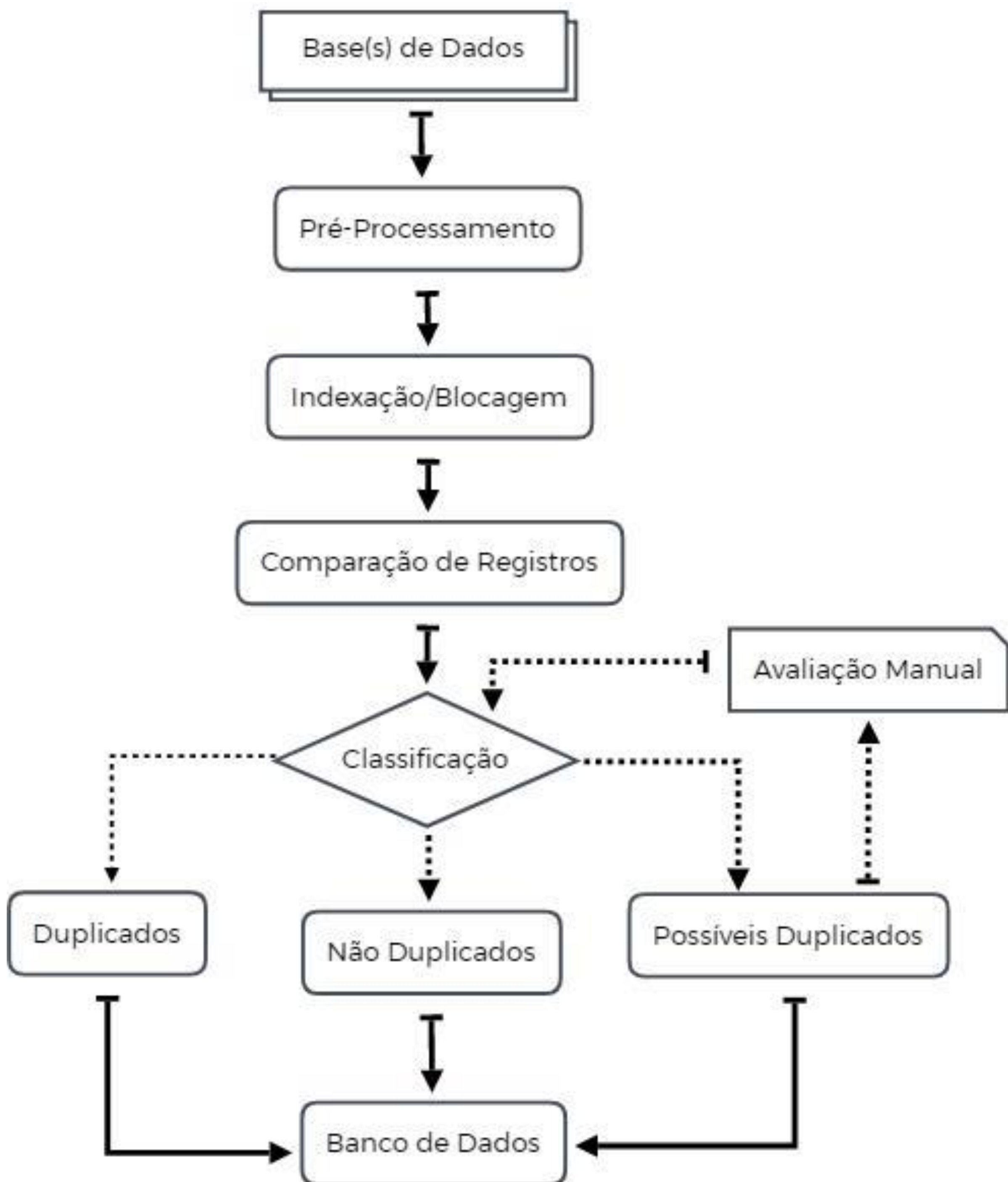
duplicidades e, assim, melhorar a qualificação de uma base de dados. Esse processo consiste na execução de diversas etapas, conforme a figura 3 que apresenta uma visão geral do processo de deduplicação de registros, adaptado de Christen(2) e Christen(3). A deduplicação é um processo utilizado em diversas áreas (18).

Record Linkage é um assunto que é tratado desde 1959, segundo artigo de Newcombe e outros autores (19). Porém, foi em 1969 que Fellegi e Sunter apresentaram um modelo com um rigor estatístico elaborado que perdura até hoje como uma das técnicas mais utilizadas no processo de deduplicação ou record linkage.

Existem modelos que utilizam técnicas de aprendizagem de máquina para execução do processo de deduplicação (20). Além desses modelos com uso de linguagem de máquina, existem os modelos que utilizam técnicas de programação genética para deduplicação (21).

Um dos benefícios da aplicação das técnicas de deduplicação é o incremento da qualidade dos registros em uma base de dados, economia por redução de espaço de armazenamento em disco, pois o volume de registros é reduzido, maior confiabilidade na extração de relatórios. Contudo, a deduplicação é um processo que demanda tempo de processamento e recurso computacional preparado, devido ao grande volume de comparações necessárias para execução do processo. Portanto, o uso de técnicas adequadas às características do banco de dados a ser deduplicado e de soluções que consumam recursos de forma eficiente são requisitos fundamentais para o sucesso do processo de deduplicação.

Figura 3 – Processo de deduplicação de registros, adaptado de Christen(2) e Christen(3).



Adaptado de Peter Christen (2009) e Peter Christen (2012)

2.3.1 Pré-Processamento

Como primeira etapa do processo exposto na figura acima, o pré-processamento tem como objetivo principal padronização dos registros. Essa etapa foca no formato,

ou forma, dos registros. Para que o processamento computacional tenha eficiência, é necessário que os registros a serem comparados tenham um tratamento com o intuito de padronizá-los. Isso aumenta, significativamente, a hipótese de classificação como duplicados de registros que não estão padronizados. Essa etapa é fundamental para o sucesso do processo de deduplicação.

O pré-prcessamento é executado, essencialmente, em três etapas, conforme lista a seguir.

1) Substituição de caracteres ou palavras com exclusão dos desnecessários;
2) "Tokenização" dos atributos dos registros a serem comparados;
3) Segmentação de atributos do registro;

Na etapa 1) os caracteres ou palavras que são desnecessárias ao processo são removidas neste procedimento. Alguns exemplos de ações de substituição ou remoção de palavras ou caracteres são apresentados na tabela 2.

Tabela 2 – Exemplos de substituições e exclusões de caracteres ou palavras no processo de pré-processamento.

Substituição	Exclusão
Caracteres acentuados por não Acentuados	Caracteres especiais como "*", "-", "/", "_", "(", ")", "?", "!" e etc.
Abreviações por palavras completas	Espaços excedentes
Caracteres numéricos por sua forma por extenso	Termos fora do contexto da comparação

Na etapa 2) o processo de "tokenização" consiste em reduzir uma expressão ou uma palavra em pequenas unidades de informação denominadas "tokens". Os tokens, neste contexto, são considerados a menor unidade de informação a ser utilizada em uma análise sintática de expressão ou termo. Por exemplo, dado uma palavra, esta é dividida em tokens e cada token é pesquisado em uma ou mais tabelas de pesquisa, deneominadas look-up. As tabelas de look-up são compostas por conjunto de palavras mapeadas como corretas, contendo suas variações. Essas variações podem ser abreviações, apelidos, sigla ou erros de tipografia. Além disso, na tabela de look-up podem ser incluídas uma ou mais etiquetas ("tags") para a palavra correta. Essas etiquetas são utilizadas na etapa de segmentação do processo de pré-processamento. No processo de padronização de um termo ou expressão com uso de tokens, cada token é pesquisado no mapeamento presente na tabela de look-up, ou seja, procura-se por

variações do termo pesquisado. A palavra padronizada é o retorno desse procedimento e a mesma substitui o termo pesquisado.

A etapa 3), segmentação, é necessária para obtenção de melhores resultados na etapa de comparação, tendo em vista que reduz o esforço computacional para a comparação dos atributos de um par de registros. Isso se deve ao fato da comparação utilizar as etiquetas ao invés dos termos ou expressões. Como as etiquetas são unidades menores de informação, o processamento ganha desempenho. Devido a grande quantidade de comparações efetuadas no processo de deduplicação, é importante lançar mão de técnicas que reduzam o tempo de processamento dessas comparações.

Existem diversas técnicas de segmentação, as duas principais são a segmentação baseada em regras e a abordagem estatística (22).

2.3.2 Indexação / Blocagem

A blocagem, segunda etapa do processo de deduplicação, é responsável, em alguns casos como bases de dados que contenham milhões de registros, como é o caso da base do Cartão Nacional de Saúde, por tornar viável, ou exequível, o processo de deduplicação. O seu objetivo principal é a redução do número de comparações necessárias, ou seja, reduzir o número de pares de registros a serem comparados. Essa, assim como o pré-processamento, é uma etapa fundamental para a viabilidade e o sucesso do processo de deduplicação.

Na blocagem, os conjuntos de registros são particionados em blocos, mutuamente excludentes e projetados com intuito de aumentar o número de casos de duplicidades (matchs). As comparações são restritas a pares de registros dentro de cada bloco (18). Com isso, reduz-se o número de comparações, tendo em vista que não é necessário comparar o registro com todo o universo do conjunto.

A separação ou agrupamento de registros em blocos é feita através de um atributo identificador que é a composição de um, ou mais atributos do registro. Esse atributo identificador é denominado chave de bloco (22). Essa composição não precisa ser do valor integral de um atributo. Ele pode contribuir na criação da chave com parte do seu valor. Por exemplo, os três primeiro dígitos do CEP do endereço de residência. O critério para escolha dessa chave é fundamental para a qualidade dos blocos criados.

> No processo de blocagem, a escolha da chave de bloco (atributo identificador) é uma das atividades mais importantes, pois a chave, aliada a técnica de blocagem empregada, será responsável pelo sucesso do processo, que é a distribuição dos registros similares em seus blocos. (22)

O número de comparações necessárias, sem uso de blocagem, em uma base

de dados com 10.000 (dez mil) registros é dado pela equação:

$$NComparacoes = \frac{n(n - 1)}{2} \quad (2.1)$$

, onde “n” é o número de registros da base de dados que serão comparados. Com isso, uma base de dados com, apenas, 10.000 (dez mil) registros terá, aplicando-se a equação acima, 49.995.000 (quarenta e nove milhões, novecentos e noventa e cinco mil) comparações para identificação de registros similares. Já uma base de dados como a do Cartão Nacional de Saúde que possuía, durante a escrita deste trabalho, cerca de 300.000.000 (trezentos milhões) de registros, esse valor de comparações seria, praticamente inviável, dado o tempo para execução. Com uso de blocagem na base de dados do primeiro exemplo, suponhamos que fossem criados 10 (dez) blocos com 1000 (um mil) registros cada, o número de comparações seria reduzido a 4.995.000 (quatro milhões, novecentos e noventa e cinco mil) comparações. Isso representaria uma redução de 90% do número de comparações.

2.3.3 Comparação

Após a etapa de blocagem, inicia-se o procedimento de comparação de cada par de registros dos blocos gerados. Para cada par de atributo dos registros é aplicada uma função de similaridade para determinar o quão similar um atributo é do outro.

A função de similaridade pode ser representada por $s = sim(ai; aj)$, onde ai e aj representam os valores de um determinado atributo para um par de registro comparado, o “sim” representa a função de similaridade e o “s” é o valor resultante da função de similaridade. Os valores resultantes de “s” estão no intervalo entre 0 e 1, ou seja, 0 s 1. Quanto maior o valor, maior o grau de similaridade. Quando o valor de “s” for 0, indica que não há qualquer similaridade entre os atributos, quanto “s” for igual a 1, indica que os valores são iguais.

Diversas são as funções de similaridades, conforme lista apresentada por Christen(23) , dentre elas Soundex, Phonex, Phonix, NYSIIS, DMetaphone, FuzSoundex, Leven, Dam-L, Bag, SWater, LCS-2, LCS-3, 1-grams, 2-grams, 3-grams, Pos, Pos, Pos, Skip, Compr, Compr, Jaro, Winkler, Editex, SyllAlign e outras mais.

A escolha de qual função de similaridade depende da natureza ou tipo dos atributos a serem comparados.

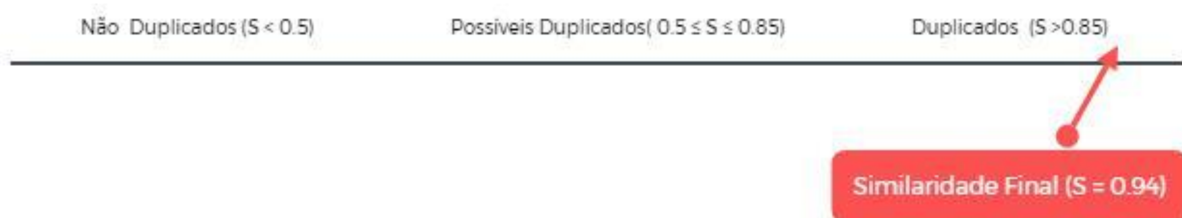
2.3.4 Classificação

A classificação é a etapa em que os registros são classificados como duplicados, não duplicados ou possíveis duplicados. A entrada dessa etapa são os registros a serem comparados com os atributos que contêm os valores de similaridade os quais

foram calculados e atribuídos na etapa de comparação. Na etapa de classificação é definido um peso final de similaridade entre os registros que é calculado com base nos valores de similaridade dos atributos.

A classificação como duplicados, não duplicados ou possíveis duplicados depende de uma definição dos valores ou intervalos de valores em que o peso final de similaridade se encontrará. A figura XX ilustra um exemplo em que foram definidos os intervalos de classificação da Similaridade Final (S) como Não Duplicados (S < 0.5), Possíveis Duplicados (0.5 S 0.85) e Duplicados (S > 0.85). A similaridade S, do exemplo, foi de 0.94. Como S > 0.85, os registros seriam classificados como Duplicados.

Figura 4 – Exemplo de Classificação com base nos intervalos definidos



Autoria própria

A classificação pode ser feita com base em dois grupos de abordagens: classificação supervisionada e não supervisionada. Diversos são os algoritmos para classificação (22). Um dos métodos de classificação mais importantes é o de Fallegi e Sunter (24).

2.3.5 Avaliação

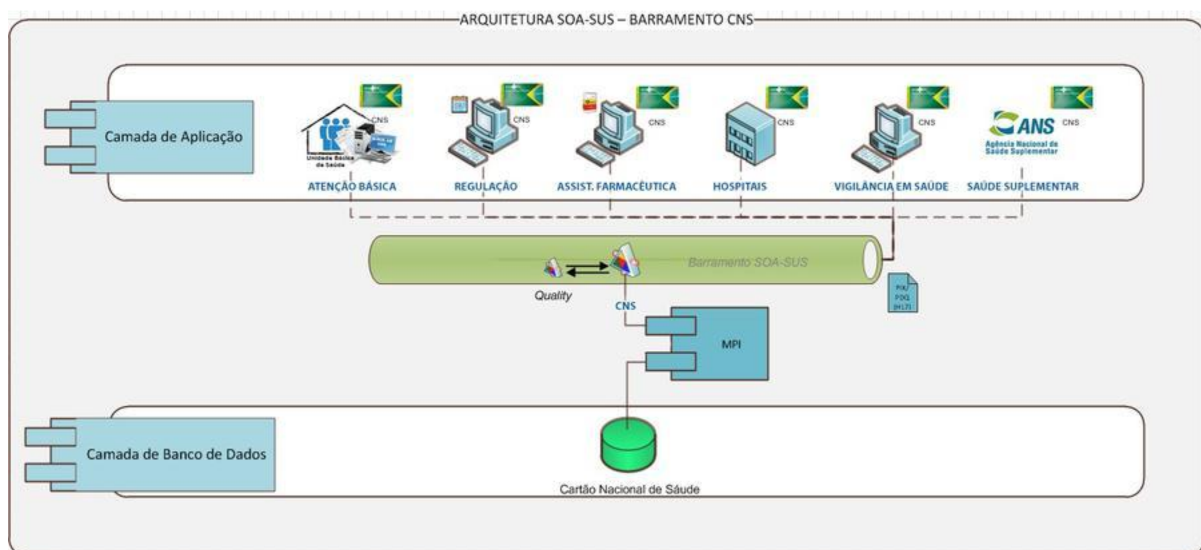
A avaliação dos resultados consiste em analisar indicadores como pair completeness, reduction ratio, accuracy, precision, recall e f-measure, por exemplo. A avaliação é um processo importante para verificar a necessidade de atualização dos processos de pré-processamento e blocagem, além de avaliar a efetividade das funções de similaridade e classificação adotadas no processo de deduplicação.

3 O Processo de deduplicação no Cartão Nacional do SUS

Como já descrito nas seções 1, 2.2 e 2.2.1 deste documento, o sistema Cartão Nacional do Saúde, regulamentado pela portaria MS nº 940, de 28 de abril de 2011, é uma base de identificação dos indivíduos que tiveram algum contato com serviços do Sistema Único de Saúde. O sistema de informação mantenedor dessas informações é denominado CADSUS e desde sua primeira versão, em 1999, até o momento de escrita deste trabalho, atualmente na versão 6 com a implantação do SGP e uma arquitetura baseada em serviços, SOA, e disponibilização de um barramento de serviços para que os estabelecimentos de saúde, mesmo os que tem sistema próprio, possam consumir os serviços do Cartão Nacional de Saúde.

A figura 5 é uma representação, em alto nível, da arquitetura do barramento SOA do sistema Cartão Nacional de Saúde (25).

Figura 5 – Arquitetura SOA do Barramento do Cartão Nacional de Saúde (CNS)



Fonte: http://datasus.saude.gov.br/inter/642-barramento-do-cns

A arquitetura evidenciada na figura 5 apresenta um componente denominado MPI. Essa sigla refere-se ao termo, em inglês, Master Patient Index, algo que, numa tradução livre, seria Identificador Mestre do Paciente.

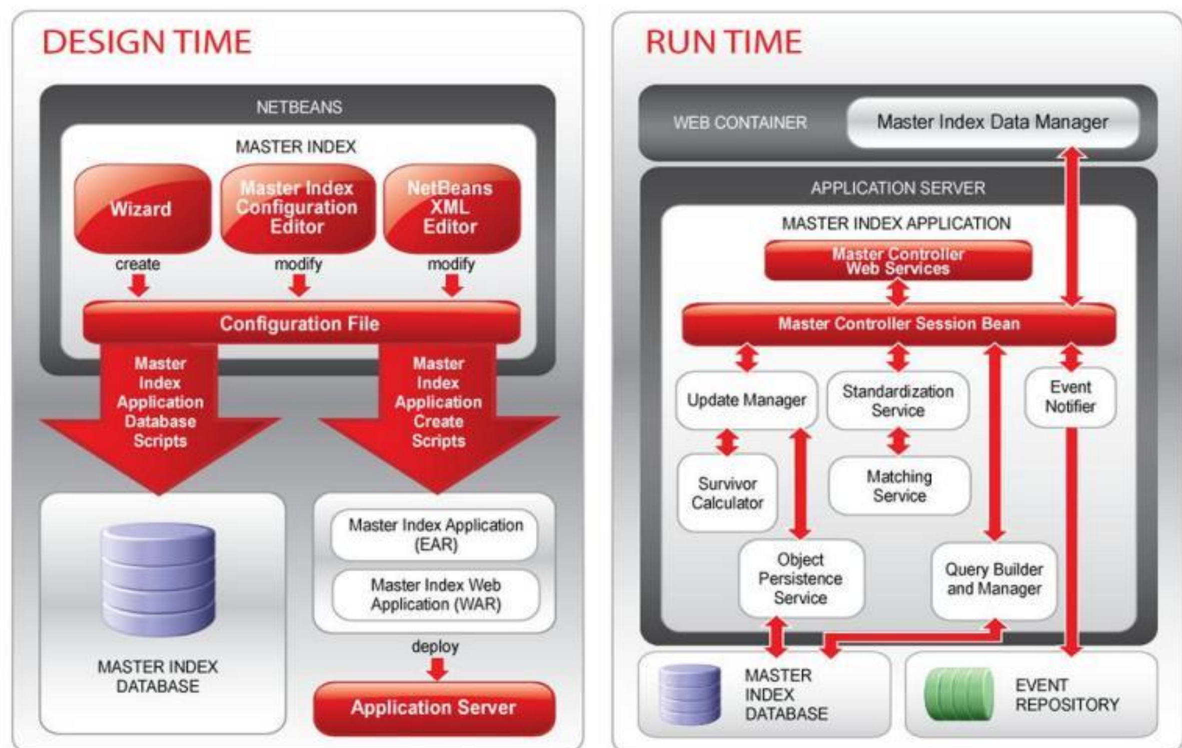
Um MPI tem como objetivo centralizar em um repositório, que indexa todos os registros de pacientes de um determinado sistema de informação. Esse repositório contempla os dados demográficos para identificação de uma pessoa. Uma solução de MPI permite sincronismo entre todos os intervenientes, minimizando inconsistências, defasagem por desatualização e redução do número de registros duplicados (26).

Para a arquitetura SOA do Cartão Nacional de Saúde, a solução de MPI implantada foi a Oracle® Healthcare Master Person Index (OHMPI) (16). A adoção de uma

solução de MPI é um dos desdobramentos do projeto de reformulação do CNS. Um dos principais objetivos de seu uso foi a redução de duplicidades nessa base nacional.

A figura 6 representa uma visão do funcionamento do OHMPI utilizado. Uma descrição mais detalhada desses componentes pode ser encontrado em (27).

Figura 6 – Representação do OHMPI em tempo de configuração e operação



Identity Resolution and Data Quality Algorithms for Master Person Index - Oracle White Paper

No caso do CNS, os campos utilizados na operação de deduplicação, ou mais especificamente de MPI, realizada no OHMPI são Nome, Data de Nascimento, Nome da Mãe, Sexo e Município de Nascimento e CPF. Esses campos são utilizados para a análise probabilística de matching. O atributo CPF, apesar de ser uma chave com identificação unívoca, devido às características negociais do CNS de não im-pedir o atendimento por ausência de informações, não é informação obrigatória no cadastramento de pessoas no CNS.

Fontes de dados externas ou de sistemas internos do Ministério da Saúde são utilizadas para melhorar o processo de deduplicação do Cartão Nacional de Saúde. Algumas delas são listadas na tabela 3. Essa tabela apresenta uma amostra das fontes externas ao CNS, ou seja, nem todas as fontes estão listadas.

Tabela 3 – Fontes de dados utilizadas na manutenção do MPI do CNS

Fonte de Dados	Sigla	Órgão Responsável/Mantenedor
Cadastro de Pessoa Física	CPF	Receita Federal do Brasil/Serpro
Sistema de Informação de Mortalidade	SIM	Ministério da Saúde/Datasus
Sistema de Controle de Óbitos	SISOBI	INSS/Dataprev
Sistema de Informações sobre Nascidos Vivos	SINASC	Ministério da Saúde/Datasus

Autoria própria

No momento da escrita deste trabalho, os algorítimos e limites configurados no OHMPI para a classificação, conforme processo descrito na seção 2.3.4, estão apresentados na figura 7.

Figura 7 – Configuração de match do OHMPI no Cartão Nacional de Saúde.

```
matchConfigFile.cfg
1  ProbabilityType        1
2
3  Sexo                   1    0    c     0.9   0.5    1   -8
4  DtNascimento           10   0    ddc   0.9   0.001  10  -20   0.85   0.65   yyyy-MM-dd
5  MunicipioNascimento    10   0    dmc   0.9   0.001  5   -5    0      2      0.4
6  CPF-BL                 12   d6   c     0.99  0.001  80  -54
7  CPF                    12   0    c     0.99  0.001  80  -54
8  CNS                    15   0    c     0.9   0.5    12  0
9  Nome                   100  0    bnac  0.9   0.001  17  -7    true   0.5    0.55   -0
10 NomeMae                100  0    bnac  0.9   0.001  17  -7    true   0.5    0.30   -0
11 NomeNaoFonetico        100  0    bnsc  0.9   0.001  1   0
12 PartoGemelar 5 2 pgc 0 0 0 -10
```

Datasus/MS

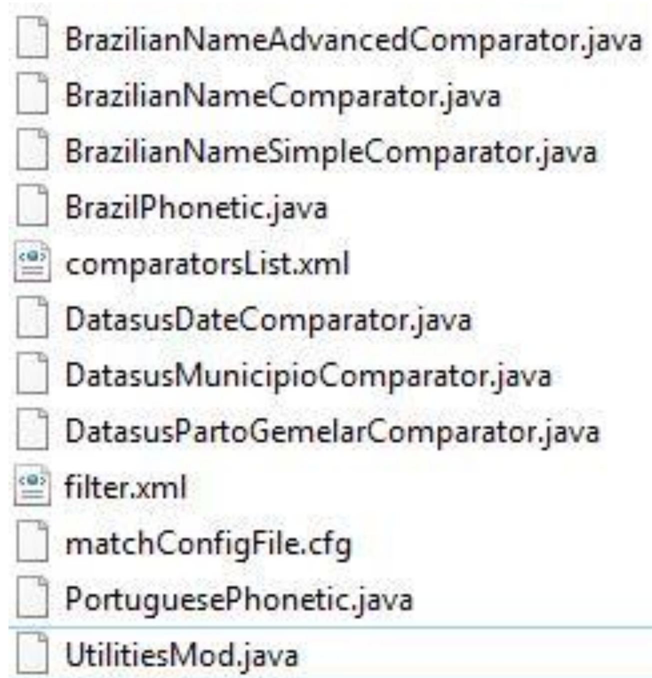
O arquivo de configuração (matchConfigFile.cfg), exposto acima, é fundamental para execução do processo de deduplicação dos registros no banco de dados do Cartão Nacional de Saúde. Nele estão as definições de quais algorítimos serão utilizados como função de similaridade da etapa de Comparação de valores de atributos, com seus limites e pesos, além de outras configurações que podem ser detalhadas no material de apoio do OHMPI (28).

O OHMPI atribui um valor inteiro para cada atributo utilizado na etapa de Comparação, eles podem ser negativos ou positivos, sem restrição de escala. Esses valores estabelecem o nível de semelhança entre os valores dos atributos comparados. Na etapa de Classificação, todos os valores atribuídos são somados e comparados aos limites para classificação. No caso do Cartão Nacional de Saúde, conforme documen-tação de regras de negócio, os limites estabelecidos são 30 para o inferior e 41 para o superior. Com isso, classificações de registros que resultarem em um valor abaixo de

30 pontos são caracterizados como não duplicados, entre 30 e 41 são classificados como possíveis duplicados e acima de 41 pontos são considerados duplicados. Os registros classificados como possíveis duplicados necessitam de uma análise manual dos usuários operadores do sistema CADSUS.

A configuração dos parâmetros para deduplicação da base do CNS considera o campo CPF, Nome, Nome da Mãe e Data de Nascimento com maior pontuação ou peso. O campo com menor influência é Sexo, seguido por Município de Nascimento. Para implantação do OHMPI, no contexto do Brasil, foi necessário criar algorítimos ou lógicas específicas para a comparação de campos específicos como a comparação de Nome, Nome da Mãe, Data de Nascimento e Município de Nascimento, conforme evidencia a figura 8. Os códigos-fontes dessas classes e arquivos de configuração listados na figura 8 e que contém os algorítmos de comparação utilizados no Cartão Nacional de Saúde estão disponíveis no Apêndice 1 deste documento.

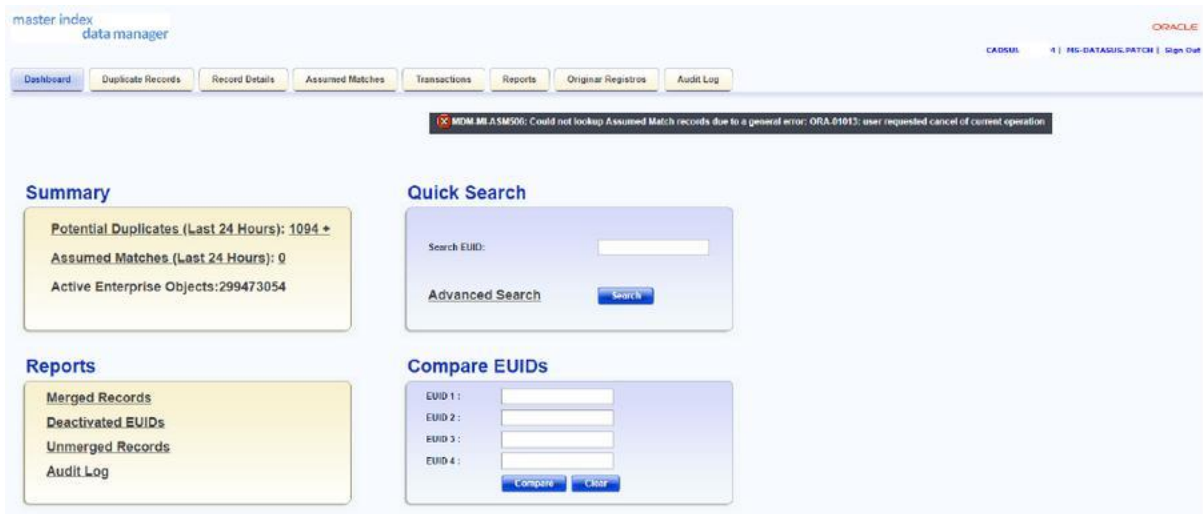
Figura 8 – Rotinas específicas para comparação de atributos de pessoa no CNS



Datasus/MS

Uma interface WEB é disponibilizada no sistema CADSUS para que os operadores possam executar o processo de validação manual dos registros classificados como possíveis duplicados. Além disso, no momento do cadastro de uma pessoa o serviço de MPI verifica, com base nas informações preenchidas se o registro já está na base do CNS, mantida pelo CADSUS. A figura 9 apresenta a tela inicial do OHMPI implantado no Ministério da Saúde.

Figura 9 – Tela inicial do OHMPI implantado no Ministério da Saúde para tratar a base de dados do CNS.



CADSUS/MS

4 Resultado da análise e discussão

Nesta seção, serão apresentados os resultados da execução de análises (caracterizações/profiling) do banco de dados do Cartão Nacional de Saúde, mantido pelo Sistema de Informação CADSUS. Essa análise foi executada com a criação de um profile na ferramenta de data quality denominada Analyst[1] da Informatica®. Foi consi-derado, neste trabalho, o modelo de dados mantido por esse sistema a partir de sua versão 5. Nessa versão do sistema CADSUS, houve a implementação do barramento com incorporação de uma solução de MPI. O profile criado no Analyst foi executado em momentos distintos para permitir avaliar a qualidade das informações no referido banco de dados ao longo de um período de tempo. Especificamente, esse profile foi executado em novembro de 2017 a fevereiro de 2019.

Para aumentar a assertividade na análise do volume estimado de duplicidades, foi levantada a informação do volume estimado da população brasileira. O Brasil, segundo a estimativa do IBGE, publicada por meio da Resolução n.º 2 de 28 de agosto de 2018 (29) , possuía, na data referência de 1 de julho de 2018, cerca de **208.494.900 (duzentos e oito milhões, quatrocentos e noventa e quatro mil e novecentos) habitantes**. Esse número foi uma referência na discussão dos resultados.

A análise da qualidade dos dados dos atributos utilizados no processo de duplicação do CNS considerou os critérios de qualidade completude, acurária/precisão e consistência. Existem outros critérios de qualidade, contudo os mesmos não serão avaliados neste trabalho.

4.1 Completude

A tabela 4 apresenta o volume de completude dos campos utilizados no processo de deduplicação. Os cinco atributos, nome, nome da mãe, data de nascimento, sexo e município de nascimento tem um alto grau de completude. Com exceção do município de nascimento, todos os outros atributos apresentaram completude de 100%. Na comparação entre a caracterização de 2017 e 2019 houve um crescimento de 3.348.948 (três milhões, trezentos e quarenta e oito mil, novecentos e quarenta e oito) de cadastros de Pessoa no banco do CNS. Para o campo município de nascimento, dado a diferença de completude, esse valor foi de 3.162.976 (três milhões cento e sessenta e dois mil e novecentos e setenta e seis) registros com esse atributo preenchido.

Esse aumento do volume de registros representa uma taxa de crescimento de 0.89% do número de registros na base do CNS entre 06/11/2017 e 19/02/2019. Uma informação importante é que o IBGE indica que entre 2017 e 2018, a população

[1] Informatica Data Quality - https://www.informatica.com/br/products/data-quality/informatica-data-qualit y.html.

brasileira aumentou a uma taxa de crescimento de 0.82% (30). A comparação entre essas duas taxas, considerando uma pequena diferença no período de apuração dos valores, evidencia que o crescimento populacional brasileiro e do número de novos registros no banco de dados do CNS, foram equivalentes. Esse achado evidencia que o tratamento de novas inclusões que é efetuado pela ferramenta OHMPI tem sido satisfatório, tendo em vista que o número de registros não foi ampliado em proporções divergentes do crescimento populacional brasileiro, levantado pelo IBGE.

Tabela 4 – Completude dos campos dematch da entidade PESSOA nas caracterizações de 2017 e 2019

Completude dos atributos utilizados no processo de duplicação							
Data Carac-Teorização (profiling)	**Nº Registros**	**Atributos utilizados no match - Número de Registros Preenchidos**					
		Nome	**Nome da Mãe**	**Data de Nasci-mento**	**Sexo**	**Município Nasci-mento**	**CPF**
06/11/2017	296.141.988	296.141.988	296.141.988	296.141.988	296.141.988	295.842.935	210.210.658
19/02/2019	299.490.936	299.490.936	299.490.936	299.490.936	299.490.936	299.005.911	220.798.578
Diferença (Valor 2019 - 2017)	3.348.948	3.348.948	3.348.948	3.348.948	3.348.948	3.162.976	10.587.920

4.2 Acurácia / Precisão

Essa seção registra a análise da acurácia ou precisão dos valores dos atributos de match elencados na seção 4.1. Cabe salientar que serão avaliadas as precisões dos valores dos tributos da base de dados, cuja caracterização foi executada em 19/02/2019. Diferente da análise de completude, a análise da acurária/precisão não necessita de uma comparação entre as caracterizações executadas em 2017 e 2019.

4.2.1 Nome

O nome é um atributo com elevada distribuição de valores. A tabela 5 evidencia que o nome com maior número de ocorrências é MARIA JOSE DA SILVA, com 82.655 (oitenta e dois mil seiscentos e cinquenta e cinco) ocorrências, representando 0.03% dos registros da base de dados. Esse é um valor pequeno, o que mostra que os registros de nomes são bem diversificados e distribuídos. Essa é uma característica relevante em processo de deduplicação.

Tabela 5 – Distribuição dos valores do atributo NOME

Nome	Nº Registros	Porcentagem
MARIA JOSE DA SILVA	82.655	0,03
MARIA APARECIDA DA SILVA	62.030	0,02
MARIA JOSE DOS SANTOS	38.352	0,01
JOSE CARLOS DA SILVA	36.871	0,01
MARIA DE LOURDES DA SILVA	35.897	0,01
MARIA APARECIDA DOS SANTOS	30.619	0,01
JOAO BATISTA DA SILVA	27.921	<0.01
JOSE ANTONIO DA SILVA	27.490	<0.01
JOSE PEREIRA DA SILVA	27.275	<0.01
MARIA DE FATIMA DA SILVA	26.609	<0.01
JOSE CARLOS DOS SANTOS	23.876	<0.01
JOSE DOS SANTOS	22.367	<0.01
JOSE FRANCISCO DA SILVA	22.057	<0.01
MARIA DO CARMO DA SILVA	21.238	<0.01
JOSE FERREIRA DA SILVA	20.050	<0.01
. . .	. . .	. . .

Na análise foram encontrados registros que não refletem ou não representam o nome verdadeiro de um indivíduo. São exemplos de nomes encontrados na base de dados do CNS: SEM NOME, NOME, PACIENTE SEM IDENTIFICAÇAO, SEM INFORMACAO, SEM DADOS PARA INFORMAR e PACIENTE USUARIO. O volume dessas ocorrências é baixo, cerca de dezenas de registros para cada um.

4.2.2 Nome da Mãe

Esse atributo possui preenchimento em todos os registros de pessoa do CNS. O valor com maior ocorrência para esse atributo é “SEM INFORMAÇÃO” que está presente em 3.912.971 (três milhões novecentos e doze mil novecentos e setenta e um) de registros, representando 1,31% da base de dados. A tabela 6 evidencia registros com Nome da Mãe inválido, inexistente ou desconhecida. O montante de registros com essas características chegou a 4.250.773 (quatro milhões duzentos e cinquenta mil e setecentos e setenta e três), equivalente a 0.7% do total de registros

da base do CNS.

Tabela 6 – Distribuição dos valores do atributo NOME DA MAE

Nome da Mãe	Nº Registros	Porcentagem
SEM INFORMAÇÃO	3.912.971	1,31
NÃO CONSTA DO REGISTRO CIVIL	90.178	0,03
MAE DESCONHECIDA	66.475	0,02
SEM INFORMAÃ!Ã¿O	32.020	0,01
SEM MAE	24.979	<0.01
MARIA	22.558	<0.01
NADA CONSTA	17.084	<0.01
SEM INFORMAÃÃO	9.439	<0.01
MAE MAE	8.992	<0.01
SEM INF	6.427	<0.01
INDISPONIVEL NO CADASTRO	4.732	<0.01
NÃO CONSTA NO REGISTRO CIVIL	4.561	<0.01
INFORMA	4.389	<0.01
SEM NOME	3.717	<0.01
SEM REGISTRO	3.503	<0.01
INF	3.248	<0.01
NOME INVALIDO OU INCOMPLETO	2.964	<0.01
SEM INFORMAÇAO	2.212	<0.01
MAE DESC	2.125	<0.01
IDENTIFICADA	1.939	<0.01
MARIA A	1.789	<0.01

Nome da Mãe	Nº Registros	Porcentagem
NOME DA MAE DO BENEFICIARIO	1.748	<0.01
NOME DA MAE	1.735	<0.01
NÃO INFORMADO	1.723	<0.01
SEM INFORMA ÃO	1.687	<0.01
IG	1.519	<0.01
SERA AJUSTADO	1.460	<0.01
FALTA NOME DA MAE	1.440	<0.01
OPERADORA DE PLANOS ODONTOLOGI	1.419	<0.01
SEM NOME DA MAE	1.372	<0.01
NC	1.359	<0.01
SEM CADASTRO	1.232	<0.01
NAO CONSTA DO REGISTRO CIVIL	1.100	<0.01
A A	990	<0.01
SEM ESPECIFICAÇÃO	988	<0.01
CONFORME CONSTA NO REGISTRO CIVIL	983	<0.01
A ESCLARECER	949	<0.01
JOSEFA	947	<0.01
DES CONHECIDA	925	<0.01
N I	895	<0.01
Total de Registros	**4.250.773**	**0.7%**

4.2.3 Data de Nascimento

A data de nascimento é um atributo preenchido em todos os registros do banco de dados do CNS. Alguns pontos chamam atenção, um deles é o fato de existirem 446.555 (quatrocentos e quarenta e seis mil quinhentos e cinquenta e cinco) registros de pessoas com data de nascimento abaixo do ano 1900. Outro é a existência de 147 (cento e quarenta e sete) registros com a data de nascimento superior à data em que a caracterização foi efetuada, ou seja, registros incluídos com data de nascimento posterior ao dia 19/02/2019.

4.2.4 Sexo

Sexo é um atributo textual que recebe os seguintes valores possíveis: M - Masculino, F - Feminino e I - Indeterminado. Esse atribuo não possui valores nulos e não há registros com valores distintos dos possíveis descritos acima. Sua distribuição está na tabela 7, que evidencia uma distribuição igualitária entre pessoas do sexo feminino e masculino na base de dados do CNS.

Tabela 7 – Distribuição dos valores do atributo SEXO

Sexo	Quantidade	Percentual
M	147.030.711	49.09%
F	152.416.328	50.89%
I	43.882	0.01%

4.2.5 Município de Nascimento

Na tabela de pessoa do CNS o dado de município de nascimento pode ser encontrado em dois atributos: MUNICIPIODENASCIMENTO e NOMEMUNICIPIONAS-CIMENTO. O primeiro tem como valores o código do IBGE, com seis dígitos. Já o segundo armazena o nome do município de nascimento, ou seja, é um campo textual.

Um ponto importante é que existe um valor que não é um padrão do IBGE, por questão de segurança, o valor dessa chave identificadora de município que re-presenta a descrição ’INVALIDO’ não será exposta. Contudo, a informação relevante para esse trabalho é que essa chave está preenchida em 74.611.668 (setenta e quatro milhões, seiscentos e onze mil e seiscentos e sessenta e oito) de registros. Isso equivale a 24.9% da base do CNS com a chave do valor ’INVALIDO“ para o campo MUNICIPIODENASCIMENTO.

O campo NOMEMUNICIPIONASCIMENTO tem como valores preenchidos como “INVALIDO” o volume de 83.530.427 (oitenta e três milhões, quinhentos e trinta mil, quatrocentos e vinte e sete). Esse valor corresponde a 27,89% com informação do campo NOMEMUNICIPIONASCIMENTO preenchida com o valor ’INVALIDO“.

Além dos registros com valor inválido, haviam registros com valor NULO para o atributo MUNICIPIODENASCIMENTO, com 485.025, o que representa 0.16% de nulos para este campo, e NOMEMUNICIPIONASCIMENTO, com 545.690, o que representa 0.18% de nulos para este, no universo do total de registros da base.

Tabela 8 – Consolidação dos valores inválidos e nulos para os atributos que representam o Município de Nascimento da Pessoa no banco do CNS

Atributo/Valores	MUNICIPIODENASCIMENTO	NOMEMUNICIPIONASCIMENTO
Inválidos	74.611.668	83.530.427
Nulos	485.025	545.690
Total	75.096.693	84.076.117
% Total	25.07%	28,07%

Autoria própria

A tabela 8 cerca de 1/4 dos registros estão com valores inválidos ou nulos para o Município de Nascimento. Outro fator que chama atenção é o da base do CNS possuir dois atributos para indicar a mesma informação. Além disso, eles não apresentaram o mesmo grau de completude, o que pode acarretar inconsistência na hora de trabalhar com os dados de município de nascimento.

4.2.6 CPF

O CPF é o atributo com maior influência na operação de deduplicação do CNS. Esse atributo apresentou uma completude de, cerca de, 74%, presente em 220.798.578 de registros da base de dados. Por se tratar de um campo identificador numérico, cuja atribuição é unívoca, não foram encontrados registros fora do padrão esperado de 11 caracteres, que o formam.

Dessa forma, a precisão, considerando os preenchidos, desse atributo é considerada satisfatória. Porém, existe um volume de 78.692.358 (setenta e oito milhões, seiscentos e noventa e dois mil, trezentos e cinquenta e oito) de registros sem a informação do CPF na base de dados do Cartão Nacional de Saúde.

4.3 Discussão

Para reforçar a discussão é importante elencar alguns números encontrados:

- O volume de 79.958.336 (setenta e nove milhões, novecentos e cinquenta e oito mil, trezentos e trinta e seis) representa o volume de 27% de duplicados, conforme registros citados na introdução deste trabalho (6) (7).
- A base de Dados do Cartão Nacional de Saúde possuía, em 19/02/2019, o volume de 299.490.936 (duzentos e noventa e nove milhões, quatrocentos e noventa mil e novecentos e trinta e seis) de registros de pessoas, conforme 4.1.

- Segundo o IBGE, população brasileira em 01/07/2018 era de 208.494.900 (duzentos e oito milhões, quatrocentos e noventa e quatro mil e novecentos) habitantes (29) .

Considerando o volume da base de dados do CNS e o volume da população, informado pelo IBGE, a diferença entre esses números resulta em 90.996.036 (noventa milhões, novecentos e noventa e seis mil e trinta e seis) registros. Na base do CNS existe um atributo booliano cujo objetivo é indicar se a pessoa representada está viva ou não. Na caracterização do dia 19/02/2019 o volume de registros com indicação de óbito representou um volume de 12.602.540 (doze milhões, seiscentos e dois mil, quinhentos e quarenta) de pessoas falecidas registradas no CNS. Como o número do IBGE considera apenas a população viva, esse volume de óbitos deve ser descontado da diferença do entre número de pessoas do CNS e do IBGE. Portanto, o resultado final dessa diferença representa 78.393.496 (setenta e oito milhões, trezentos e no-venta e três mil, quatrocentos e noventa e seis) de pessoas a mais registradas na base do cartão nacional, comparadas ao volume populacional estimado da população nacional. Esse número representa 26,16% de possíveis duplicados na base do CNS.

A hipótese apresentada na introdução deste trabalho, foi de que, com a adoção de uma ferramenta de MPI, esperava-se que o banco de dados do Cartão Nacional de Saúde tivesse um incremento na qualidade, com redução do volume de possíveis dupli-cidades. Esse problema foi relatado em diversas referências bibliográficas presentes neste documento. Concluiu-se que, em uma análise simples e direta, a adoção de uma ferramenta de MPI não resultou em melhoria no volume de possíveis duplicados na base do Cartão Nacional de Saúde, não confirmando-se a hipótese levantada na intro-dução deste trabalho. Contudo, existem ressalvas quanto a essa situação, conforme será discutido a seguir.

A conclusão acima deve levar em consideração outros fatores, ou achados, que foram encontrados nesse trabalho. Um dos achados é sobre a semelhança da taxa de crescimento da população, estimado pelo IBGE, e a taxa de crescimento do volume de registros da base do CNS. A primeira, considerando o período até julho de 2018, apresentou uma taxa de, cerca de, 0.84% no crescimento da população nacional. Já a segunda, considerando o período de novembro de 2017 a fevereiro de 2019, apresentou uma taxa aproximada de 0,89% de crescimento de registros. Considerando as diferenças entre os períodos, essas taxas podem ser consideradas semelhan-tes. Essa situação pode indicar que a base do CNS não está crescendo de forma desordenada com o uso da solução de MPI do CADSUS, implantada a partir da versão 5, comparando-se à versões anteriores que não possuíam tal mecanismo.

O segundo achado é sobre a relação do volume de CPF’s vazios na base

de dados do CNS e o volume resultante da diferença entre o total de registros de pessoas não falecidas do CNS e o volume populacional do IBGE. No CNS foram identificados 78.692.358 (setenta e oito milhões, seiscentos e noventa e dois mil, trezentos e cinquenta e oito) de registros sem a informação do CPF e na diferença acima um volume de 78.393.496 (setenta e oito milhões, trezentos e noventa e três mil, quatrocentos e noventa e seis). Esse é um volume, praticamente, igual. Com isso, pode-se inferir, dadas as devidas proporções e necessidade de análise mais profunda, que o volume de registros de pessoas duplicados na base de dados do CNS pode estar relacionada aos registros que não foram deduplicados tendo como referência os registros da base da Receita Federal do Brasil (RFB) que representa as pessoas com CPF vinculado. Um dos possíveis desdobramentos dessa situação é que o OHMPI precisa de uma configuração mais ajustada para que trate com mais acurácia os casos de comparação de registros que não possuem CPF preenchido.

Com relação à qualidade dos dados do banco de dados do Cartão Nacional de Saúde, especificamente para os atributos configurados para a deduplicação, há um atributo que merece uma atenção no que diz respeito a sua completude. Trata-se do Município de Nascimento. Essa é uma informação com cerca de 25% da base de dados sem essa informação prenchida com um município válido, ou seja, 75.096.693 (setenta e cinco milhões, noventa e seis mil, seiscentos e noventa e três) pessoas sem município de nascimento preenchido. Essa é uma informação que está presente na base de Pessoa Física da Receita Federal do Brasil. Uma solução é fazer com que essa informação seja compartilhada com o Ministério da Saúde, por meio de aditivo contratual, pois a troca de informação atual já está em um contrato, mas não há compartilhamento da informação de município de nascimento do registro da pessoa fornecido pela RFB.

5 Conclusão

A hipótese apresentada na introdução deste trabalho, foi de que, com adoção de uma ferramenta de MPI, esperava-se que o banco de dados do Cartão Nacional de Saúde (CNS) tivesse um incremento na qualidade, com redução do volume de possíveis duplicidades, problema relatado em várias referências bibliográficas presentes neste trabalho. Algumas referências ((6);(7) registraram que a base de dados do CNS possuía em torno de 27% de duplicidades. Esse fato foi confirmado com a comparação do volume de registros de pessoas, não falecidas, da base de dados do CNS com o volume populacional publicado pelo IBGE (29).

A conclusão foi de que o volume de duplicados permaneceu em um volume equivalente a 26%, mesmo após a implantação de uma solução de Master Patient Index (MPI) na arquitetura do Sistema CADSUS, em sua versão 5. Porém, com uma análise da taxa de crescimento da base de dados do CNS e da população brasileira indicada pelo IBGE, evidenciou-se que a solução tem tratado a inserção de novos registros. Com isso, o crescimento desordenado do número de registros duplicados que foi apresentado em versões anteriores do CADSUS foi solucionado. Contudo, há um volume de registros, provavelmente inseridos na operação dessas versões anteriores à 5 que o OHMPI não conseguiu tratar da forma como está configurado.

Esse trabalho contribuiu como uma fonte de informações detalhadas sobre a base de dados do Cartão Nacional de Saúde, mantida pelo sistema CADSUS, e seu processo de deduplicação. Durante a elaboração desse trabalhou percebeu-se que há pouco material relacionado ao processo de deduplicação do CNS e que traga um detalhamento como o apresentado neste trabalho.

A qualificação dessa base de dados é fundamental para a execução da estratégia e-Saúde e a existência de material que apoie ou trate desse assunto é importante para sua difusão e fomento.

Como próximos passos ou trabalhos futuros, propõe-se a execução das seguintes ações:

1) Aumentar a completude de campos como Município de Nascimento na base do CNS com recuperação de informações de outras bases de dados governamentais como a base do Cadastro de Pessoas Físicas da Receita Federal do Brasil.

2) Revisar os pesos ou limites e os algorítimos configurados no OHMPI que são utilizados na etapa de comparação com objetivo de incrementar a acurácia do processo e aumentar o volume de matchs em suas operações.

3) Com a disponibilização dos arquivos que contém os algoritmos e configurações utlizadas no processo de deduplicação do Cartão Nacional de Saúde, possibilita à comunidade acadêmica a avaliação da eficiência dos mesmos permitindo que o surgimento de novas formas ou melhorias para o processo sucitado.

4) Avaliar a viabilidade de adoção de outras abordagens como o uso de programação genética como uma alternativa para apoiar na redução desses 27% de registros duplicados ou não tratados que representam uma disparidade entre a base de dados do CNS e da população nacional estimada pelo IBGE.

Referências

1 MINISTÉRIO DA SAÚDE. **Cartão SUS - O Brasil com Saúde**. 1. ed. Brasília/DF: Ministério da Saúde, 2010. 288 p.

2 CHRISTEN, P. Development and user experiences of an open source data cleaning, deduplication and record linkage system. **SIGKDD Explorations**, v. 11, n. 1, p. 39 – 48, 2009. Disponível em: http://doi.acm.org/10.1145/1656274.1656282. Acesso em: 06/01/2019.

3 CHRISTEN, P. **Data Matching**: Concepts and Techniques for Record Linkage, Entity Resolution, and Duplicate Detection. 1. ed. [S.l.]: Springer Science & Business Media, 2012. 272 p.

4 BRASIL. **Constituição da República Federativa do Brasil**, Brasília-DF, 1988. Disponível em: http://www.planalto.gov.br/ccivil_03/constituicao/ constituicaocompilado.htm Acesso em: 06/01/2019.

5 MATTA, G. C. Princípios e Diretrizes do Sistema Único de Saúde. In: MATTA, G. C. (ed.). **Políticas de Saúde**: a organização e a operacionalização do sistema único de saúde. Rio de Janeiro: EPSJV/FIOCRUZ, 2007. p. 61 – 79.

6 MAGALHÃES, M. de A. **Desafios da Gestão de uma Base de Dados de Identifi-cação Unívoca de Indivíduos**: a experiência do Projeto Cartão Nacional de Saúde no SUS. 2010. 107 p. Dissertação (Mestrado) — Escola Nacional de Saúde Pública Sergio Arouca. Disponível em: https://www.arca.fiocruz.br/bitstream/icict/2336/1/ENSP_ Dissertaç~ao_Magalh~aes_Marcelo_de_Araujo.pdf. Acesso em: 24/02/2019.

7 BRASIL. TRIBUNAL DE CONTAS DA UNIÃO. TC 032.238/2011-8. Relatório de Levantamento - Cartão Nacional de Saúde. Brasília, 2011. Disponível em: https://www.google.com.br/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1& cad=rja&uact=8&ved=2ahUKEwiXqfiO39rgAhUPIbkGHUnRCF0QFjAAegQIBxAC& url=https%3A%2F%2Fcontas.tcu.gov.br%2Fetcu%2FObterDocumentoSisdoc% 3FseAbrirDocNoBrowser%3Dtrue%26codArqCatalogado%3D4258996&usg= AOvVaw3RmeEAVxxzEnk_ZpNtvOsN. C.

8 BRASIL - Constituição Federal 88. 1988. Disponível em: http://www.planalto.gov. br/ccivil_03/constituicao/constituicaocompilado.htm. Acesso em: 18/02/2019.

9 MINISTÉRIO DA SAÚDE. **Sistema Único de Saúde (SUS): estrutura, princípios e como funciona**. Brasília/DF: [s.n.], 2019. Disponível em: http://portalms.saude.gov. br/sistema-unico-de-saude. Acesso em: 18/02/2019.

10 SAITO, R. X. de S. et al. **Sistema Único de Saúde**. 2004. Dissertação (Mestrado) — Universidade de São Paulo. Disponível em: http://www.teses.usp.br/teses/disponiveis/7/7139/tde-28062007-100306/ . Acesso em: 18/02/2019.

11 BRASIL, MINISTÉRIO DA SAÚDE. ESTRATÉGIA e-SAÚDE PARA O BRA-SIL. 2017. Disponível em: http://www.conasems.org.br/wp-content/uploads/2019/ 02/Estrategia-e-saude-para-o-Brasil.pdf . Acesso em: 26/02/2019.

12 BRASIL, COMISSÃO INTERGESTORES TRIPARTITE. Institui o Comitê Gestor da Estratégia e-Saúde e define a sua composição, competência, funcionamento e

unidades operacionais na estrutura do Ministério da Saúde. **Resolução nº 5, DE 25 DE AGOSTO DE 2016**, Brasília, Agosto 2016. Disponível em: http://www.conass.org. br/wp-content/uploads/2016/12/RESOLUCAO-N_5_16.pdf . Acesso em: 24/02/2019.

13 ANDRADE, M. V. et al. CAPÍTULO 26 - DESAFIOS DO SISTEMA DE SAÚDE BRASILEIRO. In: ANDRADE, M. V. et al. (ed.). **DESAFIOS DA NAÇÃO**: Artigos de apoio, volume 2. Brasília: nstituto de Pesquisa Econômica Aplicada (Ipea), 2018. v. 2, cap. 26. ISBN 978-85-7811-322-3. Disponível em: http://repositorio.ipea.gov.br/ handle/11058/8468 . Acesso em: 24/02/2019.

14 CONASS. Nota Técnica 22/2011. **PROPOSTA DE CONSOLIDAÇÃO DO CARTÃO NACIONAL DE SAÚDE**, Brasília, Junho 2011. Disponível em: http://www.conass.org.br/biblioteca/wp-content/uploads/2011/01/NT-22_2011_projeto_consolidacao_cartao_sus_atualizacao.pdf Acesso em: 24/02/2019

15 DATASUS/MS. **CADSUS**. Brasília: [s.n.], 2019. Portal. Disponível em: <http:// datasus.saude.gov.br/sistemas-e-aplicativos/cadastros-nacionais/cadsus>. Acesso em: 24/02/2019.

16 ORACLE. **Oracle Healthcare Master Person Index (OHMPI)**. 2019. Disponível em: http://www.oracle.com/us/products/applications/health-sciences/master-person/overview/index.html. Acesso em: 24/02/2019.

17 DATASUS/MS. **Sistema de Gestão de Operadores (SGOP)**. 2019. Disponível em: http://datasus.saude.gov.br/sistemas-e-aplicativos/cadastros-nacionais/cadsus/sgop. Acesso em: 24/02/2019.

18 HERZOG, T. N.; SCHEUREN, F. J.; WINKLER, W. E. **Data Quality and Record Linkage Techniques**. 1. ed. Verlag New York: Springer, 2007. 234 p.

19 NEWCOMBE, H. et al. Automatic linkage of vital records. **Science**, n. 130, p. 954 – 959, 1959. Disponível em: https://www.cs.umd.edu/class/spring2012/cmsc828L/ Papers/Newcombe59.pdf. Acesso em: 25/02/2019.

20 BILENKO, M. et al. Adaptive Name Matching in Information Integration. **IEEE Intelligent Systems**, v. 18, n. 5, p. 16 – 23, 2003. Disponível em: http://doi. ieeecomputersociety.org/10.1109/MIS.2003.1234765.

21 GONCALVES, G. S. **Seleção automática de exemplos de treino para um mé-todo de deduplicação de registros baseado em programação genética**. 2010. 79 p. Dissertação (Ciência da Computação) — Universidade Federal de Minas Ge-rais. Disponível em: http://www.bibliotecadigital.ufmg.br/dspace/bitstream/handle/ 1843/BUBD-9JWQAQ/dissertacao_gabrielsilvagoncalves.pdf?sequence=1. C.

22 CAVALIERI, O. M. et al. **Um método complementar ao processo de sanitiza-ção de registros duplicados em bases de dados Cadsus-multiplataforma**. 2014. Dissertação (Mestrado). Disponível em: http://dspace.c3sl.ufpr.br:8080/dspace/ handle/1884/36297. Acesso em: 18/02/2019.

23 CHRISTEN, P. A comparison of personal name matching: Techniques and practice al issues. In: IEEE (Ed.). **Proceedings of the Sixth IEEE International Conference on**

Data Mining Workshops, ICDMW ’06. Washington, DC, USA: IEEE Computer Society, 2006. p. 290 – 294.

24 FELLEGI, I. P.; SUNTER, A. B. A theory for record linkage. **Journal of the Ame-rican Statistical Society**, v. 64, n. 328, 1969. Disponível em: https://courses.cs.washington.edu/courses/cse590q/04au/papers/Felligi69.pdf . Acesso em: 24/02/2019.

25 DATASUS/MS. **Barramento do CNS**. Brasília/DF: [s.n.], 2014. Disponível em: http://datasus.saude.gov.br/inter/642-barramento-do-cns. Acesso em: 05/03/2019.

26 GOMES, P. T. M. **Master Patient Index**. 2009. 83 p. Dissertação (Engenharia de Redes e Sistemas Informáticos) — Faculdade de Ciências Universidade do Porto. Disponível em: http://www.dcc.fc.up.pt/~ptmgomes/documents/MSc_PedroGomes. pdf. Acesso em: 05/03/2019.

27 ORACLE. **Identity Resolution and Data Quality Algorithms for Master Person Index**. USA, 2010. Disponível em: http://www.oracle.com/us/industries/healthcare/ identity-resolution-algorithm-wp-171743.pdf. Acesso em: 05/03/2019.

28 ORACLE. **OHMPI - Match Engine Configuration for Common Data**. 2011. Disponível em: https://docs.oracle.com/html/E68420_01/mime_ref_chapter3.htm. Acesso em: 10/03/2019.

29 IBGE. RESOLUÇÃO Nº 2, DE 28 DE AGOSTO DE 2018. **Estimativas da População para Estados e Municípios com data de referência em 1º de ju-lho de 2018**, Brasília/DF, Agosto 2018a. Disponível em: http://pesquisa.in.gov.br/imprensa/jsp/visualiza/index.jsp?jornal=515&pagina=55&data=29/08/2018. Acesso em: 24/02/2019.

30 IBGE. **IBGE divulga as Estimativas de População dos municípios para 2018**. Brasília/DF: [s.n.], 2018b. Disponível em: https://agenciadenoticias.ibge.gov.br/ agencia-sala-de-imprensa/2013-agencia-de-noticias/releases/22374-ibge-divulga-as-estimativas-de-populacao-dos-municipios-para-2018. Acesso em: 05/03/2019.

APÊNDICES

Apêndice A

Códigos-fontes das classes que implementam os algoritmos de comparação no processo de deduplicação do Cartão Nacional de Saúde:

BrazilianNameAdvancedComparator.java

Código 1 – BrazilianNameAdvancedComparator.java

```
1 package com.sun.mdm.matcher.comparators.addon;
2
3 import com.sun.mdm.matcher.comparators.MatchComparator;
4 import com.sun.mdm.matcher.comparators.MatchComparatorException;
5 import java.text.MessageFormat;
6 import java.util.ArrayList;
7 import java.util.Arrays;
8 import java.util.Collections;
9 import java.util.HashMap;
10 import java.util.HashSet;
11 import java.util.LinkedList;
12 import java.util.List;
13 import java.util.Map;
14 import java.util.Map.Entry;
15 import java.util.Set;
16 import java.util.TreeSet;
17
18 public class BrazilianNameAdvancedComparator implements
       MatchComparator {
19
20     private HashSet<String> IGNORED_WORDS = new HashSet<String>();
21     private Map<String, Map> params;
22     private Map<String, String> theParams = null;
23     private Map<String, String> argumentsRT = new HashMap<String,
           String>();
24
25     @Override
26     public void initialize(Map<String, Map> params, Map<String, Map
           > dataSources, Map<String, Map> dependClassList) {
27         this.params = params;
28         IGNORED_WORDS.addAll(Arrays.asList("DU", "DA", "DI", "I", "
               NAU", "NU", "KU", "A"));
29     }
30
31     @Override
32     public void setRTParameters(String key, String value) {
33         this.argumentsRT.put(key, value);
34     }
35
36     @Override
37     public void stop() {
```

```
38          this.argumentsRT.clear();
39      }
40
41      @Override
42      public double compareFields(String str1, String str2, Map
            context) throws MatchComparatorException {
43
44          this.theParams = ((Map) this.params.get(context.get("
                fieldName")));
45
46          boolean ignoreCommonNames = false;
47          double commonWordsNotSameOrderPenaltyMultiplier = 0.0;
48          double wordDistanceAcceptance = 0.0;
49          double firstWordNotComparedPenalty = 0.0;
50
51          try {
52
53              ignoreCommonNames = Boolean.parseBoolean(this.theParams
                    .get("ignoreCommonNames"));
54              commonWordsNotSameOrderPenaltyMultiplier = Double.
                    parseDouble(this.theParams.get("
                    commonWordsNotSameOrderPenaltyMultiplier"));
55              wordDistanceAcceptance = Double.parseDouble(this.
                    theParams.get("wordDistanceAcceptance"));
56              firstWordNotComparedPenalty = Double.parseDouble(this.
                    theParams.get("firstWordNotComparedPenalty"));
57
58          } catch (Exception e) {
59              throw new MatchComparatorException(
60                      "One of these params is invalid:
                            ignoreCommonNames,
                            commonWordsNotSameOrderPenaltyMultiplier,
                            wordDistanceAcceptance,
                            firstWordNotComparedPenalty");
61          }
62
63          if (str1 == null || str2 == null || (str1.trim().length()
                == 0) || (str2.trim().length() == 0)) {
64              return 0;
65          }
66
67          /*
68           *  calculate
69           */
70
71          List<String> allTokens1 = new ArrayList<String>();
72          Collections.addAll(allTokens1, str1.split("\\s+"));
73
74          List<String> allTokens2 = new ArrayList<String>();
75          Collections.addAll(allTokens2, str2.split("\\s+"));
76
77          if(ignoreCommonNames){
78              allTokens1 = cleanseCommonNames(allTokens1);
79              allTokens2 = cleanseCommonNames(allTokens2);
80          }
81
82          /*
83           * remove duplicates and alphabetic ordenate the tokens
84           */
```

```
85        Set<String> setTokens1 = new HashSet<String>();
86        setTokens1.addAll(allTokens1);
87
88        Set<String> setTokens2 = new HashSet<String>();
89        setTokens2.addAll(allTokens2);
90
91        int length1 = setTokens1.size();
92        int length2 = setTokens2.size();
93
94        /*
95         *  intersection
96         */
97        Set<String> intersection = new HashSet<String>();
98        intersection.addAll(setTokens1);
99        intersection.retainAll(setTokens2);
100
101       int commonWordLength = intersection.size();
102
103       /*
104        * if the common words are in same order in two Strings
              match weight is
105        * 1.0 else match weight is penalized
106        */
107       List<String> commonTokensOrdered1 = new LinkedList<String
             >();
108       List<String> commonTokensOrdered2 = new LinkedList<String
             >();
109
110       for (String token : allTokens1) {
111           if (intersection.contains(token)) {
112               commonTokensOrdered1.add(token);
113           }
114       }
115       for (String token : allTokens2) {
116           if (intersection.contains(token)) {
117               commonTokensOrdered2.add(token);
118           }
119       }
120
121       boolean sameOrder = true;
122       for (int i = 0; i < commonWordLength; i++) {
123           if (!commonTokensOrdered1.get(i).equals(
                 commonTokensOrdered2.get(i))) {
124               sameOrder = false;
125               break;
126           }
127       }
128
129       /*
130        * compare word by word
131        */
132       HashSet<Pair> mapFinal = new HashSet<Pair>();
133
134       Set<String> tempSetTokens1 = new HashSet<String>();
135       tempSetTokens1.addAll(setTokens1);
136
137       Set<String> tempSetTokens2 = new HashSet<String>();
```

```
138         tempSetTokens2.addAll(setTokens2);
139
140         while (tempSetTokens1.size() > 0) {
141
142             HashMap<String, TreeSet<Pair>> mapTemp = new HashMap<
                    String, TreeSet<Pair>>();
143             Set<String> repetedTokens = new HashSet<String>();
144
145             for (String token1 : tempSetTokens1) {
146
147                 double maxMatch = -1.0;
148                 String maxMatchIndex = "";
149                 for (String token2 : tempSetTokens2) {
150
151                     double d = calculateDistance(token1, token2);
152                     if (d > maxMatch) {
153                         maxMatch = d;
154                         maxMatchIndex = token2;
155                     }
156                 }
157
158                 Pair pair = new Pair(token1, maxMatchIndex,
                        maxMatch);
159
160                 if(mapTemp.containsKey(maxMatchIndex)){
161                     // token2 already used
162                     TreeSet<Pair> pairs = mapTemp.get(maxMatchIndex
                            );
163                     pairs.add(pair);
164                     repetedTokens.add(maxMatchIndex);
165
166                 } else {
167                     // token2 not used yet
168                     TreeSet<Pair> pairs = new TreeSet<Pair>();
169                     pairs.add(pair);
170                     mapTemp.put(maxMatchIndex, pairs);
171                 }
172             }
173
174             for (Entry<String, TreeSet<Pair>> entry : mapTemp.
                    entrySet()) {
175
176                 /*
177                  * get always the first from the treeset, that one
                        will be the
178                  * single on using that token2, or the one with the
                         biggest
179                  * match value using the same token2 (because the
                        set is sorted
180                  * by the match value)
181                  */
182                 Pair pair = entry.getValue().first();
183                 mapFinal.add(pair);
184
185                 tempSetTokens1.remove(pair.token1);
186                 tempSetTokens2.remove(pair.token2);
187             }
```

```
188         }
189 
190         /*
191          * apply  wordDistanceAcceptance  and  sum
192          */
193         String firstWord1 = allTokens1.size() > 0 ? allTokens1.get
               (0) : null;
194         String firstWord2 = allTokens2.size() > 0 ? allTokens2.get
               (0) : null;
195         boolean applyFirstWordPenalty = false;
196 
197         double  sum  = 0.0;
198         for (Pair pair : mapFinal) {
199 
200             // first word from name 1 was not compared to
201             // first word from name 2
202             if (pair.token1.equals(firstWord1) && !pair.token2.
                   equals(firstWord2)) {
203                 applyFirstWordPenalty = true;
204             }
205 
206             //apply wordDistanceAcceptance
207             if (pair.weight < wordDistanceAcceptance) {
208                 pair.weight = 0.0;
209             }
210 
211             sum += pair.weight;
212         }
213 
214         // se fizer a união dos tokens dos dois nomes,
215         // o tamanho do conjuntos será length1 + length2 -
               commonWordLength
216         double res = sum / (double) (length1 + length2 -
               commonWordLength);
217 
218         // apply First Word Penalty
219         if (applyFirstWordPenalty) {
220             res += firstWordNotComparedPenalty;
221         }
222 
223         if (!sameOrder) {
224             res = res * commonWordsNotSameOrderPenaltyMultiplier;
225         }
226 
227         return  res;
228     }
229 
230     private double calculateDistance(String str1, String str2) {
231 
232         int[] costs = new int[str2.length() + 1];
233 
234         for (int i = 0; i <= str1.length(); i++) {
235 
236             int lastValue = i;
237             for (int j = 0; j <= str2.length(); j++) {
238 
239                 if (i == 0) {
```

```
240                         costs[j] = j;
241                     } else {
242 
243                         if (j > 0) {
244 
245                             int newValue = costs[j - 1];
246                             if (str1.charAt(i - 1) != str2.charAt(j -
                                    1)) {
247                                 newValue = Math.min(Math.min(newValue,
                                        lastValue), costs[j]) + 1;
248                             }
249 
250                             costs[j - 1] = lastValue;
251                             lastValue = newValue;
252                         }
253                     }
254                 }
255 
256                 if (i > 0) {
257                     costs[str2.length()] = lastValue;
258                 }
259             }
260 
261             double distance = 1.0 - costs[str2.length()] / (double)
                    Math.max(str1.length(), str2.length());
262             return distance;
263         }
264 
265         private List<String> cleanseCommonNames(List<String> tokens) {
266 
267             ArrayList<String> list = new ArrayList<String>();
268 
269             for (String token : tokens) {
270 
271                 if (!IGNORED_WORDS.contains(token)) {
272                     list.add(token);
273                 }
274             }
275 
276             return list;
277         }
278 }
279 
280 class Pair implements Comparable<Pair> {
281 
282         protected String token1;
283         protected String token2;
284         protected Double weight = 0.0;
285 
286         Pair(String token1, String token2, Double weight) {
287             this.token1 = token1;
288             this.token2 = token2;
289             this.weight = weight > 1.0 ? 1.0 : weight < 0.0 ? 0.0 :
                    weight;
290         }
291 
292         @Override
```

```
293     public int hashCode() {
294
295         final int prime = 31;
296         int result = 1;
297         result = prime * result + ((token1 == null) ? 0 : token1.
               hashCode());
298         result = prime * result + ((token2 == null) ? 0 : token2.
               hashCode());
299         result = prime * result + ((weight == null) ? 0 : weight.
               hashCode());
300         return result;
301     }
302
303     @Override
304     public boolean equals(Object obj) {
305
306         if (this == obj) {
307             return true;
308         }
309         if (obj == null) {
310             return false;
311         }
312         if (getClass() != obj.getClass()) {
313             return false;
314         }
315
316         Pair other = (Pair) obj;
317         if (token1 == null) {
318             if (other.token1 != null) {
319                 return false;
320             }
321         } else if (!token1.equals(other.token1)) {
322             return false;
323         }
324         if (token2 == null) {
325             if (other.token2 != null) {
326                 return false;
327             }
328         } else if (!token2.equals(other.token2)) {
329             return false;
330         }
331         if (weight == null) {
332             if (other.weight != null) {
333                 return false;
334             }
335         } else if (!weight.equals(other.weight)) {
336             return false;
337         }
338
339         return true;
340     }
341
342     @Override
343     public int compareTo(Pair other) {
344         // reversed order
345         return other.weight.compareTo(this.weight);
346     }
```

```
347
348     @ Override
349     public  String  toString () {
350         return MessageFormat.format("[{0}, {1}, {2}]", token1 ,
              token2 , weight);
351     }
352 }
```

BrazilianNameComparator.java

Código 2 – BrazilianNameComparator.java

```
 1 /**
 2  * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.
 3  *
 4  * Copyright 2003-2007 Sun Microsystems, Inc. All Rights Reserved.
 5  *
 6  * The contents of this file are subject to the terms of the Common
 7  * Development and Distribution License ("CDDL")(the "License").
      You
 8  * may not use this file except in compliance with the License.
 9  *
10  * You can obtain a copy of the License at
11  * https://open-dm-mi.dev.java.net/cddl.html
12  * or open-dm-mi/bootstrap/legal/license.txt. See the License for
      the
13  * specific language governing permissions and limitations under
      the
14  * License.
15  *
16  * When distributing the Covered Code, include this CDDL Header
      Notice
17  * in each file and include the License file at
18  * open-dm-mi/bootstrap/legal/license.txt.
19  * If applicable, add the following below this CDDL Header, with
      the
20  * fields enclosed by brackets [] replaced by your own identifying
21  * information: "Portions Copyrighted [year] [name of copyright
      owner]"
22  */
23 package com.sun.mdm.matcher.comparators.addon;
24
25 import com.sun.mdm.index.phonetic.impl.PortuguesePhonetic;
26 import java.util.Map;
27 import com.sun.mdm.matcher.comparators.MatchComparator;
28 import com.sun.mdm.matcher.comparators.MatchComparatorException;
29 import java.util.HashSet;
30 import java.util.LinkedList;
31 import java.util.Set;
32 import java.util.logging.Level;
33 import java.util.logging.Logger;
34
35 /**
36  * @author pkar
```

```
37  */
38 public class BrazilianNameComparator implements MatchComparator {
39
40     /**
41      * Initialize the parameters and data sources info.
42      * @param params provides all the parameters associated with
           a given match field using this match comparator
43      * @param dataSources provides all the data sources info.
           associated with a given match field using this match
           comparator
44      * @param dependClassList provides the list of all the
           dependency classes
45      */
46     public void initialize(Map<String, Map> params, Map<String, Map
         > dataSources, Map<String, Map> dependClassList) {
47     }
48
49     /**
50      * A setter for real-time passed-in parameters
51      *
52      * @param key the key for use in a Map
53      * @param value the corresponding value for use in a Map
54      */
55     public void setRTParameters(String key, String value) {
56     }
57
58     /**
59      * Reads two strings and measure how close they are relying on
           an algorithm
60      * that compare the proximity of the two strings (zero being
           very different and
61      * one being identical)
62      *
63      * @param recordA Candidate's string record.
64      * @param recordB Reference's string record.
65      * @return a real number between zero and one that measures
           the degree of similarity.
66      */
67     public double compareFields(String str1, String str2, Map
         context)
68             throws MatchComparatorException {
69
70         //return simpleCommonWord(str1, str2);
71         return multiWordEditDistance(str1, str2);
72     }
73
74     private double simpleCommonWord(String str1, String str2) {
75
76         double prob;
77
78         if (str1 == null || str2 == null || (str1.trim().length()
             == 0) || (str2.trim().length() == 0)) {
79             return 0;
80         }
81         String[] tokens1 = str1.split("\\s+");
82         String[] tokens2 = str2.split("\\s+");
83         int length1 = tokens1.length;
```

```
84          int length2 = tokens2.length;
85          HashSet<String> HS1 = new HashSet<String>();
86          HashSet<String> HS2 = new HashSet<String>();
87          for (int i = 0; i < length1; i++) {
88              HS1.add(tokens1[i]);
89          }
90          for (int i = 0; i < length2; i++) {
91              HS2.add(tokens2[i]);
92          }
93          Set<String> intersection = intersection(HS1, HS2);
94          double commonWordLength = intersection.size();
95          //System.out.println("common length " + commonWordLength);
96
97          Set<String> difference = union(HS1, HS2);
98          double unionWordLength = difference.size();
99          //System.out.println("union length " + unionWordLength);
100
101         LinkedList<String> AL1 = new LinkedList<String>();
102         LinkedList<String> AL2 = new LinkedList<String>();
103
104         //if the common words are in same order in two Strings
               match weight is 1.0
105         //else match weight is 0.5
106         for (int i = 0; i < length1; i++) {
107             if (intersection.contains(tokens1[i])) {
108                 AL1.add(tokens1[i]);
109             }
110         }
111         for (int i = 0; i < length2; i++) {
112             if (intersection.contains(tokens2[i])) {
113                 AL2.add(tokens2[i]);
114             }
115         }
116         boolean sameOrder = true;
117         for (int i = 0; i < commonWordLength; i++) {
118             if (!AL1.get(i).equals(AL2.get(i))) {
119                 sameOrder = false;
120                 break;
121             }
122         }
123         //System.out.println("Formula 1 " + commonWordLength /
               unionWordLength);
124         //System.out.println("Formula 2" + commonWordLength / (
               length1 + length2 - commonWordLength));
125         prob = commonWordLength / (length1 + length2 -
               commonWordLength);
126         if (!sameOrder) {
127             prob = prob * 0.5;
128         }
129         return prob;
130     }
131
132     private double editDistance(String str1, String str2) {
133
134         int[] costs = new int[str2.length() + 1];
135         for (int i = 0; i <= str1.length(); i++) {
136             int lastValue = i;
```

```
137             for (int j = 0; j <= str2.length(); j++) {
138                 if (i == 0) {
139                     costs[j] = j;
140                 } else {
141                     if (j > 0) {
142                         int newValue = costs[j - 1];
143                         if (str1.charAt(i - 1) != str2.charAt(j -
                            1)) {
144                             newValue = Math.min(Math.min(newValue,
                                lastValue), costs[j]) + 1;
145                         }
146                         costs[j - 1] = lastValue;
147                         lastValue = newValue;
148                     }
149                 }
150             }
151             if (i > 0) {
152                 costs[str2.length()] = lastValue;
153             }
154         }
155         return 1.0 - costs[str2.length()] / (double) max(str1.
            length(), str2.length());
156     }
157 
158     private double multiWordEditDistance(String str1, String str2)
        {
159 
160         double prob;
161 
162         if (str1 == null || str2 == null || (str1.trim().length()
            == 0) || (str2.trim().length() == 0)) {
163             return 0;
164         }
165         String[] tokens1 = str1.split("\\s+");
166         String[] tokens2 = str2.split("\\s+");
167         int length1 = tokens1.length;
168         int length2 = tokens2.length;
169 
170         HashSet<String> HS1 = new HashSet<String>();
171         HashSet<String> HS2 = new HashSet<String>();
172         for (int i = 0; i < length1; i++) {
173             HS1.add(tokens1[i]);
174         }
175         for (int i = 0; i < length2; i++) {
176             HS2.add(tokens2[i]);
177         }
178 
179         Set<String> intersection = intersection(HS1, HS2);
180         double commonWordLength = intersection.size();
181 
182         /*
183          * if the common words are in same order in two Strings
             match weight is
184          * 1.0 else match weight is 0.6
185          */
186         LinkedList<String> AL1 = new LinkedList<String>();
187         LinkedList<String> AL2 = new LinkedList<String>();
```

```
188
189         for (int i = 0; i < length1; i++) {
190             if (intersection.contains(tokens1[i])) {
191                 AL1.add(tokens1[i]);
192             }
193         }
194         for (int i = 0; i < length2; i++) {
195             if (intersection.contains(tokens2[i])) {
196                 AL2.add(tokens2[i]);
197             }
198         }
199         boolean sameOrder = true;
200         for (int i = 0; i < commonWordLength; i++) {
201             if (!AL1.get(i).equals(AL2.get(i))) {
202                 sameOrder = false;
203                 break;
204             }
205         }
206
207         double sum = 0.0;
208         for (String s1 : HS1) {
209             double maxMatch = 0.0;
210             String maxMatchString = "";
211             for (String s2 : HS2) {
212                 double d = editDistance(s1, s2);
213                 if (d > maxMatch) {
214                     maxMatch = d;
215                     maxMatchString = s2;
216                 }
217             }
218             HS2.remove(maxMatchString);
219             sum = sum + maxMatch;
220         }
221
222         //System.out.println("match weight from not common words "
               + (sum - commonWordLength));
223         double res = sum / (double) (length1 + length2 -
               commonWordLength);
224
225         if (!sameOrder) {
226             res = res * 0.6;
227         }
228
229         return res;
230     }
231
232     public <T> Set<T> union(Set<T> setA, Set<T> setB) {
233         Set<T> tmp = new HashSet<T>(setA);
234         tmp.addAll(setB);
235         return tmp;
236     }
237
238     public <T> Set<T> intersection(Set<T> setA, Set<T> setB) {
239         Set<T> tmp = new HashSet<T>();
240         for (T x : setA) {
241             if (setB.contains(x)) {
242                 tmp.add(x);
```

```
243                 }
244             }
245             return tmp;
246         }
247 
248         public <T> Set<T> difference(Set<T> setA, Set<T> setB) {
249             Set<T> tmp = new HashSet<T>(setA);
250             tmp.removeAll(setB);
251             return tmp;
252         }
253 
254         private int min(int a, int b) {
255 
256             return a < b ? a : b;
257         }
258 
259         private int max(int a, int b) {
260 
261             return a > b ? a : b;
262         }
263 
264         /**
265          * Close any related data sources streams
266          */
267         public void stop() {
268         }
269 
270         public static void main(String args[]) {
271 
272             double limiteSup = 17.0;
273             double limiteInf = -7;
274 
275             BrazilianNameComparator bnc = new BrazilianNameComparator()
                    ;
276             PortuguesePhonetic phon = new PortuguesePhonetic();
277             String a = null;
278             String b = null;
279             a = phon.CalculatePortuguesePhonetic("LUIZ ANTONIO ALVES
                    LINO E SILVA");
280             b = phon.CalculatePortuguesePhonetic("ANDRE LUIZ ALVES LINO
                     E SILVA");
281             System.out.println(a);
282             System.out.println(b);
283 
284             try {
285 
286                 double valor = bnc.compareFields(a, b, null);
287                 double valor2 = (valor * (limiteSup - limiteInf)) +
                        limiteInf;
288 
289                 System.out.println(valor);
290                 System.out.println(valor2);
291 
292             } catch (MatchComparatorException ex) {
293                 Logger.getLogger(BrazilianNameComparator.class.getName
                        ()).log(Level.SEVERE, null, ex);
294             }
```

```
295     }
296 }
```

BrazilianNameSimpleComparator.java

Código 3 – BrazilianNameSimpleComparator.java

```
1  /**
2   * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.
3   *
4   * Copyright 2003-2007 Sun Microsystems, Inc. All Rights Reserved.
5   *
6   * The contents of this file are subject to the terms of the Common
7   * Development and Distribution License ("CDDL")(the "License").
       You
8   * may not use this file except in compliance with the License.
9   *
10  * You can obtain a copy of the License at
11  * https://open-dm-mi.dev.java.net/cddl.html
12  * or open-dm-mi/bootstrap/legal/license.txt. See the License for
       the
13  * specific language governing permissions and limitations under
       the
14  * License.
15  *
16  * When distributing the Covered Code, include this CDDL Header
       Notice
17  * in each file and include the License file at
18  * open-dm-mi/bootstrap/legal/license.txt.
19  * If applicable, add the following below this CDDL Header, with
       the
20  * fields enclosed by brackets [] replaced by your own identifying
21  * information: "Portions Copyrighted [year] [name of copyright
       owner]"
22  */
23 package com.sun.mdm.matcher.comparators.addon;
24
25 import com.sun.mdm.index.phonetic.impl.PortuguesePhonetic;
26 import java.util.Map;
27 import com.sun.mdm.matcher.comparators.MatchComparator;
28 import com.sun.mdm.matcher.comparators.MatchComparatorException;
29 import java.util.HashSet;
30 import java.util.LinkedList;
31 import java.util.Set;
32 import java.util.logging.Level;
33 import java.util.logging.Logger;
34
35 /**
36  * @author pkar
37  */
38 public class BrazilianNameSimpleComparator implements
     MatchComparator {
39
40     /**
```

```
41         * Initialize the parameters and data sources info.
42         * @param  params  provides all the parameters associated with
              a given match field using this match comparator
43         * @param  dataSources  provides all the data sources info.
              associated with a given match field using this match
              comparator
44         * @param  dependClassList  provides the list of all the
              dependency classes
45         */
46        public void initialize(Map<String, Map> params, Map<String, Map
             > dataSources, Map<String, Map> dependClassList) {
47        }
48
49        /**
50         * A setter for real-time passed-in parameters
51         *
52         * @param  key  the key for use in a Map
53         * @param  value the corresponding value for use in a Map
54         */
55        public void setRTParameters(String key, String value) {
56        }
57
58        /**
59         * Reads two strings and measure how close they are relying on
              an algorithm
60         * that compare the proximity of the two strings (zero being
              very different and
61         * one being identical)
62         *
63         * @param  recordA  Candidate's string record.
64         * @param  recordB  Reference's string record.
65         * @return  a real number between zero and one that measures
              the degree of similarity.
66         */
67        public double compareFields(String str1, String str2, Map
             context)
68                throws MatchComparatorException {
69
70            return multiWordEditDistance(str1, str2);
71        }
72
73        private double editDistance(String str1, String str2) {
74
75            int[] costs = new int[str2.length() + 1];
76            for (int i = 0; i <= str1.length(); i++) {
77                int lastValue = i;
78                for (int j = 0; j <= str2.length(); j++) {
79                    if (i == 0) {
80                        costs[j] = j;
81                    } else {
82                        if (j > 0) {
83                            int newValue = costs[j - 1];
84                            if (str1.charAt(i - 1) != str2.charAt(j -
                                 1)) {
85                              newValue = Math.min(Math.min(newValue,
                                  lastValue), costs[j]) + 1;
86                            }
```

```
87                      costs[j - 1] = lastValue;
88                      lastValue = newValue;
89                  }
90              }
91          }
92          if (i > 0) {
93              costs[str2.length()] = lastValue;
94          }
95      }
96      return 1.0 - costs[str2.length()] / (double) max(str1.
           length(), str2.length());
97  }
98
99  private double multiWordEditDistance(String str1, String str2)
       {
100
101     double prob;
102
103     if (str1 == null || str2 == null || (str1.trim().length()
           == 0) || (str2.trim().length() == 0)) {
104         return 0;
105     }
106     String[] tokens1 = str1.split("\\s+");
107     String[] tokens2 = str2.split("\\s+");
108     int length1 = tokens1.length;
109     int length2 = tokens2.length;
110
111     HashSet<String> HS1 = new HashSet<String>();
112     HashSet<String> HS2 = new HashSet<String>();
113     for (int i = 0; i < length1; i++) {
114         HS1.add(tokens1[i]);
115     }
116     for (int i = 0; i < length2; i++) {
117         HS2.add(tokens2[i]);
118     }
119
120     Set<String> intersection = intersection(HS1, HS2);
121     double commonWordLength = intersection.size();
122
123     /*
124      * if the common words are in same order in two Strings
           match weight is
125      * 1.0 else match weight is 0.6
126      */
127     LinkedList<String> AL1 = new LinkedList<String>();
128     LinkedList<String> AL2 = new LinkedList<String>();
129
130     for (int i = 0; i < length1; i++) {
131         if (intersection.contains(tokens1[i])) {
132             AL1.add(tokens1[i]);
133         }
134     }
135     for (int i = 0; i < length2; i++) {
136         if (intersection.contains(tokens2[i])) {
137             AL2.add(tokens2[i]);
138         }
139     }
```

```
140         boolean sameOrder = true;
141         for (int i = 0; i < commonWordLength; i++) {
142             if (!AL1.get(i).equals(AL2.get(i))) {
143                 sameOrder = false;
144                 break;
145             }
146         }
147
148         double sum = 0.0;
149         for (String s1 : HS1) {
150             double maxMatch = 0.0;
151             String maxMatchString = "";
152             for (String s2 : HS2) {
153                 double d = editDistance(s1, s2);
154                 if (d > maxMatch) {
155                     maxMatch = d;
156                     maxMatchString = s2;
157                 }
158             }
159             HS2.remove(maxMatchString);
160             sum = sum + maxMatch;
161         }
162
163         //verify if first words are equal. if not, penalty of 0.3
164         boolean firstsEquals = true;
165         if (length1 > 0 && length2 > 0) {
166             double firstDistance = editDistance(tokens1[0], tokens2
                    [0]);
167             if (firstDistance == 1.0) {
168                 firstsEquals = true;
169             } else {
170                 firstsEquals = false;
171             }
172         }
173
174         //System.out.println("match weight from not common words "
                + (sum - commonWordLength));
175         double res = sum / (double) (length1 + length2 -
                commonWordLength);
176
177         if (!sameOrder) {
178             res = res * 0.6;
179         }
180
181         if (!firstsEquals) {
182             res = res * 0.7;
183         }
184
185         return res;
186     }
187
188     public <T> Set<T> union(Set<T> setA, Set<T> setB) {
189         Set<T> tmp = new HashSet<T>(setA);
190         tmp.addAll(setB);
191         return tmp;
192     }
193
```

```
194     public <T> Set<T> intersection(Set<T> setA, Set<T> setB) {
195         Set<T> tmp = new HashSet<T>();
196         for (T x : setA) {
197             if (setB.contains(x)) {
198                 tmp.add(x);
199             }
200         }
201         return tmp;
202     }
203
204     public <T> Set<T> difference(Set<T> setA, Set<T> setB) {
205         Set<T> tmp = new HashSet<T>(setA);
206         tmp.removeAll(setB);
207         return tmp;
208     }
209
210     private int min(int a, int b) {
211
212         return a < b ? a : b;
213     }
214
215     private int max(int a, int b) {
216
217         return a > b ? a : b;
218     }
219
220     /**
221      * Close any related data sources streams
222      */
223     public void stop() {
224     }
225
226     public static void main(String args[]) {
227
228         double limiteSup = 17.0;
229         double limiteInf = -7;
230
231         BrazilianNameSimpleComparator bnc = new
                BrazilianNameSimpleComparator();
232         PortuguesePhonetic phon = new PortuguesePhonetic();
233         String a = null;
234         String b = null;
235         a = phon.CalculatePortuguesePhonetic("LUIZ ANTONIO ALVES
                LINO E SILVA");
236         b = phon.CalculatePortuguesePhonetic("ANDRE LUIZ ALVES LINO
                 E SILVA");
237         System.out.println(a);
238         System.out.println(b);
239
240         try {
241
242             double valor = bnc.compareFields(a, b, null);
243             double valor2 = (valor * (limiteSup - limiteInf)) +
                    limiteInf;
244
245             System.out.println(valor);
246             System.out.println(valor2);
```

```
247
248             } catch (MatchComparatorException ex) {
249                 Logger.getLogger(BrazilianNameSimpleComparator.class.
                        getName()).log(Level.SEVERE, null, ex);
250             }
251         }
252 }
```

BrazilPhonetic.java

Código 4 – BrazilPhonetic.java

```
 1 /*

 2  * DO NOT ALTER OR REMOVE COPYRIGHT NOTICES OR THIS HEADER.
 3  *
 4  * Copyright 2003-2007 Sun Microsystems, Inc. All Rights Reserved.
 5  *
 6  * The contents of this file are subject to the terms of the Common
 7  * Development and Distribution License ("CDDL")(the "License").
      You
 8  * may not use this file except in compliance with the License.
 9  *
10  * You can obtain a copy of the License at
11  * https://open-dm-mi.dev.java.net/cddl.html
12  * or open-dm-mi/bootstrap/legal/license.txt. See the License for
      the
13  * specific language governing permissions and limitations under
      the
14  * License.
15  *
16  * When distributing the Covered Code, include this CDDL Header
      Notice
17  * in each file and include the License file at
18  * open-dm-mi/bootstrap/legal/license.txt.
19  * If applicable, add the following below this CDDL Header, with
      the
20  * fields enclosed by brackets [] replaced by your own identifying
21  * information: "Portions Copyrighted [year] [name of copyright
      owner]"
22  */
23 package com.sun.mdm.index.phonetic.impl;
24
25 import com.sun.mdm.index.phonetic.PhoneticEncoder;
26 import com.sun.mdm.index.phonetic.PhoneticEncoderException;
27 import com.sun.mdm.index.util.Localizer;
28
29 /**
30  * Encode a string using a version of French Soundex algorithm (
      Soundex2 by .
31  *
32  * @version $Revision: 1.1 $
33  */
34 public class BrazilPhonetic implements PhoneticEncoder {
35
```

```
36      private transient final Localizer mLocalizer = Localizer.get();
37
38      /** Informative String about the encoding type this encoder
           does */
39      public static final String ENCODING_TYPE = "Portuguese_BR";
40
41      public BrazilPhonetic() {
42          super();
43      }
44
45      public String encode(String str) throws
           PhoneticEncoderException {
46
47          PortuguesePhonetic P = new PortuguesePhonetic();
48          return(P.CalculatePortuguesePhonetic(str));
49      }
50
51      public String encode(String value, String domain) throws
           PhoneticEncoderException {
52           return (encode(value));
53      }
54
55      public String getEncodingType() {
56          return ENCODING_TYPE;
57      }
58
59 }
```

DatasusDateComparator.java

Código 5 – DatasusDateComparator.java

```
 1 package com.sun.mdm.matcher.comparators.addon;
 2
 3 import com.sun.mdm.matcher.comparators.MatchComparator;
 4 import com.sun.mdm.matcher.comparators.MatchComparatorException;
 5 import java.text.DateFormat;
 6 import java.text.MessageFormat;
 7 import java.text.ParseException;
 8 import java.text.SimpleDateFormat;
 9 import java.util.Calendar;
10 import java.util.Date;
11 import java.util.GregorianCalendar;
12 import java.util.HashMap;
13 import java.util.Locale;
14 import java.util.Map;
15
16 /**
17  *
18  *
19  */
20 public class DatasusDateComparator implements MatchComparator {
21
22      private Map<String, Map> params;
```

```
23      private Map<String, String> theParams = null;
24      private Map<String, String> argumentsRT = new HashMap<String,
           String>();
25
26      @Override
27      public void initialize(Map<String, Map> params, Map<String, Map
           > dataSources, Map<String, Map> dependClassList) {
28          this.params = params;
29      }
30
31      @Override
32      public void setRTParameters(String key, String value) {
33          this.argumentsRT.put(key, value);
34      }
35
36      @Override
37      public void stop() {
38          this.argumentsRT.clear();
39      }
40
41      @Override
42      public double compareFields(String dateA, String dateB, Map
           context) throws MatchComparatorException {
43
44          this.theParams = ((Map) this.params.get(context.get("
               fieldName")));
45
46          try {
47              Double.parseDouble(this.theParams.get("oneDiffWeight"))
                   ;
48              Double.parseDouble(this.theParams.get("twoDiffsWeight")
                   );
49          } catch (Exception e) {
50              throw new MatchComparatorException("oneDiffWeight or
                   twoDiffsWeight param invalid.");
51          }
52
53          String dateFormat;
54          if ((dateFormat = (String) this.argumentsRT.get("DateFormat
               ")) == null) {
55              if ((dateFormat = (String) this.theParams.get("
                   dateFormat")) == null) {
56                  dateFormat = "yyyyMMdd";
57              }
58          }
59
60          Locale locale;
61          if (this.argumentsRT.get("Locale") != null) {
62              locale = new Locale((String) this.argumentsRT.get("
                   Locale"));
63          } else {
64              locale = Locale.US;
65          }
66
67          DateFormat df = new SimpleDateFormat(dateFormat, locale);
68
69          Date date1 = null;
```

```
70          Date date2 = null;
71
72          synchronized (df) {
73
74              try {
75
76                  date1 = df.parse(dateA);
77                  date2 = df.parse(dateB);
78
79              } catch (ParseException e) {
80
81                  try {
82
83                      dateFormat = "yyyyMMdd";
84                      df = new SimpleDateFormat(dateFormat, locale);
85
86                      date1 = df.parse(dateA);
87                      date2 = df.parse(dateB);
88
89                  } catch (ParseException ex) {
90                      System.out.println(MessageFormat.format("format
                            ={0}, date1={1}, date2={2}", dateFormat,
                            dateA, dateB));
91                      ex.printStackTrace();
92                      throw new MatchComparatorException(ex);
93                  }
94              }
95          }
96
97          // date 1 parsing
98          Calendar cal1 = new GregorianCalendar();
99          cal1.setTime(date1);
100         String[] timeSA = new String[3];
101         timeSA[0] = String.valueOf(cal1.get(Calendar.YEAR));
102         timeSA[1] = String.valueOf(cal1.get(Calendar.MONTH) + 1);
103         timeSA[2] = String.valueOf(cal1.get(Calendar.DAY_OF_MONTH))
                ;
104
105         while (timeSA[0].length() < 4) {
106             timeSA[0] = "0" + timeSA[0];
107         }
108
109         while (timeSA[1].length() < 2) {
110             timeSA[1] = "0" + timeSA[1];
111         }
112
113         while (timeSA[2].length() < 2) {
114             timeSA[2] = "0" + timeSA[2];
115         }
116
117         // date 2 parsing
118         Calendar cal2 = new GregorianCalendar();
119         cal2.setTime(date2);
120         String[] timeSB = new String[3];
121         timeSB[0] = String.valueOf(cal2.get(Calendar.YEAR));
122         timeSB[1] = String.valueOf(cal2.get(Calendar.MONTH) + 1);
```

```
123         timeSB[2] = String.valueOf(cal2.get(Calendar.DAY_OF_MONTH))
               ;
124
125         while (timeSB[0].length() < 4) {
126             timeSB[0] = "0" + timeSB[0];
127         }
128
129         while (timeSB[1].length() < 2) {
130             timeSB[1] = "0" + timeSB[1];
131         }
132
133         while (timeSB[2].length() < 2) {
134             timeSB[2] = "0" + timeSB[2];
135         }
136
137         double weight = 0.0D;
138
139         // will compare pairs: day, month, year divided into 2
               pairs.
140         String[] pairs1 = new String[4];
141         pairs1[0] = timeSA[0].substring(0,2);
142         pairs1[1] = timeSA[0].substring(2);
143         pairs1[2] = timeSA[1];
144         pairs1[3] = timeSA[2];
145
146         String[] pairs2 = new String[4];
147         pairs2[0] = timeSB[0].substring(0,2);
148         pairs2[1] = timeSB[0].substring(2);
149         pairs2[2] = timeSB[1];
150         pairs2[3] = timeSB[2];
151
152         // how many errors?
153         int errors = 0;
154         for (int x = 0; x < 4; x++) {
155             if (!pairs1[x].equals(pairs2[x])) {
156                 errors++;
157             }
158         }
159
160         if (errors == 0) {
161             weight = 1.0;
162         } else if (errors == 1) {
163             weight = Double.parseDouble(this.theParams.get("
                   oneDiffWeight"));
164
165         } else if (errors == 2) {
166             weight = Double.parseDouble(this.theParams.get("
                   twoDiffsWeight"));
167
168         } else {
169             weight = 0.0;
170         }
171
172         return weight;
173     }
174 }
```

DatasusMunicipioComparator.java

Código 6 – DatasusMunicipioComparator.java

```
1 package com.sun.mdm.matcher.comparators.addon;
2
3 import com.sun.mdm.matcher.comparators.MatchComparator;
4 import com.sun.mdm.matcher.comparators.MatchComparatorException;
5 import java.util.HashMap;
6 import java.util.Map;
7
8 /**
9  *
10  *
11  */
12 public class DatasusMunicipioComparator implements MatchComparator
       {
13
14     private Map<String, Map> params;
15     private Map<String, String> theParams = null;
16     private Map<String, String> argumentsRT = new HashMap<String,
           String>();
17
18     @Override
19     public void initialize(Map<String, Map> params, Map<String, Map
           > dataSources, Map<String, Map> dependClassList) {
20         this.params = params;
21     }
22
23     @Override
24     public void setRTParameters(String key, String value) {
25         this.argumentsRT.put(key, value);
26     }
27
28     @Override
29     public void stop() {
30         this.argumentsRT.clear();
31     }
32
33     @Override
34     public double compareFields(String municA, String municB, Map
           context) throws MatchComparatorException {
35
36         this.theParams = ((Map) this.params.get(context.get("
               fieldName")));
37
38         try {
39
40             Integer.parseInt(this.theParams.get("
                   beginIndexParcialAgreement"));
41             Integer.parseInt(this.theParams.get("
                   endIndexParcialAgreement"));
42             Double.parseDouble(this.theParams.get("
                   parcialAgreementWeight"));
43
```

```
44          } catch (Exception e) {
45              throw new MatchComparatorException(
46                      "beginIndexParcialAgreement,
                        endIndexParcialAgreement or
                        parcialAgreementWeight params invalid.");
47          }
48
49          if (municA == null || municB == null) {
50              throw new MatchComparatorException("Forbiden comparison
                    within null values");
51          }
52
53          double weight = 0.0D;
54
55          if (municA.equals(municB)) {
56              weight = 1.0;
57
58          } else {
59
60              int beginIndex = Integer.parseInt(this.theParams.get("
                    beginIndexParcialAgreement"));
61              int endIndex = Integer.parseInt(this.theParams.get("
                    endIndexParcialAgreement"));
62
63              if (municA.substring(beginIndex, endIndex).equals(
                    municB.substring(beginIndex, endIndex))) {
64                  weight = Double.parseDouble(this.theParams.get("
                        parcialAgreementWeight"));
65              }
66
67          }
68
69          return weight;
70      }
71 }
```

DatasusPartoGemelarComparator.java

Código 7 – DatasusPartoGemelarComparator.java

```
1 /*

2  * To change this template, choose Tools | Templates
3  * and open the template in the editor.
4  */
5
6 package com.sun.mdm.matcher.comparators.addon;
7
8 import com.sun.mdm.matcher.comparators.MatchComparator;
9 import com.sun.mdm.matcher.comparators.MatchComparatorException;
10 import java.util.HashMap;
11 import java.util.Map;
12
13 /**
14  *
```

```
15  *
16  */
17 public class DatasusPartoGemelarComparator implements
      MatchComparator {
18
19     private Map<String, Map> params;
20     private Map<String, String> theParams = null;
21     private Map<String, String> argumentsRT = new HashMap<String,
          String>();
22
23     @Override
24     public void initialize(Map<String, Map> params, Map<String, Map
          > dataSources, Map<String, Map> dependClassList) {
25         this.params = params;
26     }
27
28     @Override
29     public void setRTParameters(String key, String value) {
30         this.argumentsRT.put(key, value);
31     }
32
33     @Override
34     public void stop() {
35         this.argumentsRT.clear();
36     }
37
38     @Override
39     public double compareFields(String partoGemelarA, String
          partoGemelarB, Map context) throws MatchComparatorException
          {
40
41         double weight;
42
43         if ((partoGemelarA != null && partoGemelarA.equals("true"))
              || (partoGemelarB
44                 != null && partoGemelarB.equals("true"))) {
45
46             weight = 0.0D;
47
48         } else {
49
50             weight = 1.0D;
51
52         }
53
54         return weight;
55     }
56 }
```

matchConfigFile.cfg

Código 8 – matchConfigFile.cfg

```
1  ProbabilityType        1
```

```
2
3 Sexo                     1      0      c      0.9    0.5    1
    -8
4 DtNascimento             10     0      ddc    0.9    0.001  10
    -20     0.85    0.65    yyyy-MM-dd
5 Municipio Nascimento     10     0      dmc    0.9    0.001  5
    -5      0       2       0.4
6 CPF-BL                   12     d6     c      0.99   0.001  80
    -54
7 CPF                      12     0      c      0.99   0.001  80
    -54
8 CNS                      15     0      c      0.9    0.5    12
    0
9 Nome                     100    0      bnac   0.9    0.001  17
    -7      true    0.5     0.55    -0.20
10 NomeMae                 100    0      bnac   0.9    0.001  17
    -7      true    0.5     0.30    -0.20
11 NomeNaoFonetico         100    0      bnsc   0.9    0.001  1
    0
12 PartoGemelar 5 2 pgc 0 0 0 -10
```

PortuguesePhonetic.java

Código 9 – PortuguesePhonetic.java

```
1 package com.sun.mdm.index.phonetic.impl;
2
3 import java.util.concurrent.CopyOnWriteArrayList;
4
5 public class PortuguesePhonetic {
6
7     UtilitiesMod u = new UtilitiesMod();
8
9
10   public String CalculatePortuguesePhonetic(String str) {
11
12       str = str.toUpperCase();
13       str = u.removeAccentuation(str);
14       str = u.removeStrange(str);
15       str = fonetize(str);
16       return str;
17       }
18
19   public static void main(String[] args){
20
21         PortuguesePhonetic P = new PortuguesePhonetic();
22         System.out.println(P.CalculatePortuguesePhonetic("ISABELLA
             DOS SANTOS RANGEL"));
23         System.out.println(P.CalculatePortuguesePhonetic("LUIZ
             RIBEIRO GADU XYZ"));
24         System.out.println(P.CalculatePortuguesePhonetic("LUIS RIBIRO
              GADU XYZ"));
25
26   }
```

```
27
28    private String fonetize(String str) {
29
30
31            char[] foncmp = new char[256];
32            char[] fonwrk = new char[256];
33            char[] fonaux = new char[256];
34            char[] fonfon = new char[256];
35
36            int i, j, x, k,
37                desloc,
38                endfon,
39                copfon,
40                copmud, newmud;
41
42            CopyOnWriteArrayList<String> component = new
                  CopyOnWriteArrayList<String>();
43
44            i = 0;
45            j = 0;//zera os contadores
46
47            str = u.removeMultiple(str);
48            //todos os caracteres duplicados sao eliminados
49            //exemplo: SS -> S, RR -> R
50
51            component = u.strToList(str);
52            //o texto eh armazenado no vetor:
53            //cada palavra ocupa uma posicao do vetor
54
55            desloc = 0;
56            for (String s: component) {
57            //percorre o vetor, palavra a palavra
58
59              for (i = 0; i < 256; i++) {
60                fonwrk[i] = ' ';
61                fonfon[i] = ' ';//branqueia as matrizes
62              }//for
63
64              foncmp = s.toCharArray();
65              fonaux = foncmp;
66              //matrizes recebem os caracteres da palavra atual
67
68              j = 0;
69
70              if (s.length() == 1) {
71                fonwrk[0] = foncmp[0];
72              //se a palavra possuir apenas 1 caracter, nao altera a
                    palavra
73
74                if (foncmp[0] == '_') {
75                  fonwrk[0] = ' ';
76                //se o caracter for "_", troca por espaco em branco
77
78                }// if
79                else
80                  if ((foncmp[0] == 'E')||
81                      (foncmp[0] == '&')||
```

```
82                      (foncmp[0] == 'I')) {
83                    fonwrk[0] = 'i';
84                    //se o caracter for "E", "&" ou "I", troca por "i"
85
86                  }// if
87              }// if
88              else {
89                for (i = 0; i < s.length(); i++)
90                //percorre a palavra corrente, caracter a caracter
91
92                  if (foncmp[i] == '_')
93                    fonfon[i] = 'Y';      // _ -> Y
94                  else
95                    if (foncmp[i] == '&')
96                      fonfon[i] = 'i';      //& -> i
97                    else
98                      if ((foncmp[i] == 'E') ||
99                          (foncmp[i] == 'Y') ||
100                         (foncmp[i] == 'I'))
101                       fonfon[i] = 'i';      // E, Y, I -> i
102                     else
103                       if ((foncmp[i] == 'O') ||
104                           (foncmp[i] == 'U'))
105                         fonfon[i] = 'o';      // O, U -> u
106                       else
107                         if (foncmp[i] == 'A')
108                           fonfon[i] = 'a';      // A -> a
109                         else
110                           if (foncmp[i] == 'S')
111                             fonfon[i] = 's';      // S -> s
112                           else
113                             fonfon[i] = foncmp[i];
114                             //caracter nao eh modificado
115
116               endfon = 0;
117               fonaux = fonfon;
118
119               //palavras formadas por apenas 3 consoantes
120               //sao dispensadas do processo de fonetizacao
121               if (fonaux[3] == ' ')
122                 if ((fonaux[0] == 'a') ||
123                     (fonaux[0] == 'i') ||
124                     (fonaux[0] == 'o'))
125                   endfon = 0;
126                 else
127                   if ((fonaux[1] == 'a') ||
128                       (fonaux[1] == 'i') ||
129                       (fonaux[1] == 'o'))
130                     endfon = 0;
131                   else
132                     if ((fonaux[2] == 'a') ||
133                         (fonaux[2] == 'i') ||
134                         (fonaux[2] == 'o'))
135                       endfon = 0;
136                     else {
137                       endfon = 1;
138                       fonwrk[0] = fonaux[0];
```

```
139                 fonwrk[1] = fonaux [1];
140                 fonwrk[2] = fonaux [2];
141               }//else
142
143         if (endfon != 1) { //se a palavra nao for formada por
               apenas 3 consoantes...
144           for (i = 0; i < s.length(); i++) {
145           //percorre a palavra corrente, letra a letra
146
147             copfon = 0;
148             copmud = 0;
149             newmud = 0;
150             //zera variaveis de controle
151
152             switch (fonaux[i]) {
153
154               case 'a':  //se o caracter for a
155
156                 //se a palavra termina com As, AZ, AM, ou AN,
157                 //elimina a consoante do final da palavra
158                 if ((fonaux[i+1]== 's') ||
159                     (fonaux[i+1]== 'Z') ||
160                     (fonaux[i+1]== 'M') ||
161                     (fonaux[i+1]== 'N'))
162                   if(fonaux[i+2]!= ' ')
163                     copfon = 1;
164                   else {
165                     fonwrk[j] = 'a';
166                     fonwrk[j+1] = ' ';
167                     j++;
168                     i++;
169                   }//else
170                 else copfon = 1;
171               break;
172
173               case 'B':  //se o caracter for B
174
175                 // B nao eh modificado
176                 copmud = 1;
177               break;
178
179               case 'C':  //se o caracter for C
180
181                 x = 0;
182                 if (fonaux[i+1] == 'i')
183
184                 //ci vira si
185                 { fonwrk[j] = 's';
186                   j++;
187                   break;
188                 }// if
189
190                 //coes final vira cao
191                 if ((fonaux[i+1] == 'o') &&
192                     (fonaux[i+2] == 'i') &&
193                     (fonaux[i+3] == 's') &&
194                     (fonaux[i+4] == ' '))
```

```
195             { fonwrk[j] = 'K';
196               fonwrk[j+1] = 'a';
197               fonwrk[j+2] = 'o';
198               i = i + 4;
199               break;
200             }//if
201
202             //ct vira t
203             if (fonaux[i+1] == 'T')
204               break;
205
206             //    c vira k
207             if (fonaux[i+1] != 'H')
208             { fonwrk[j] = 'K';
209               newmud = 1;
210
211               //    ck vira k
212               if (fonaux[i+1] == 'K')
213               { i++;
214                 break;
215               }//if
216
217               else break;
218             }//if
219
220             //ch vira k para chi final, chi vogal, chini
                  final e
221             //chiti final
222
223             //chi final ou chi vogal
224             if (fonaux[i+1] == 'H')
225               if (fonaux[i+2] == 'i')
226                 if ((fonaux[i+3] == 'a')||
227                     (fonaux[i+3] == 'i')||
228                     (fonaux[i+3] == 'o'))
229                   x = 1;
230
231             // chini final
232             else
233               if (fonaux[i+3] == 'N')
234                 if (fonaux[i+4] == 'i')
235                   if (fonaux[i+5] == ' ')
236                     x = 1;
237
238                   else;
239                 else;
240               else
241                 // chiti final
242                 if (fonaux[i+3] == 'T')
243                   if (fonaux[i+4] == 'i')
244                     if (fonaux[i+5] == ' ')
245                       x = 1;
246             if (x == 1)
247             { fonwrk[j] = 'K';
248               j++;
249               i++;
250               break;
```

```
251                }// if
252
253            //chi, nao chi final, chi vogal, chini final ou
                 chiti final
254            //ch nao seguido de i
255            //se anterior nao e s, ch = x
256            if (j > 0)
257
258              //sch: fonema recua uma posicao
259              if (fonwrk[j-1] == 's')
260                { j--;
261                }// if
262              fonwrk[j] = 'X';
263              newmud = 1;
264              i++;
265            break;
266
267            case 'D':  //se o caracter for D
268              x = 0;
269
270              //procura por dor
271              if (fonaux[i+1] != 'o')
272              { copmud = 1;
273                break;
274              }// if
275              else
276                if (fonaux[i+2] == 'R')
277                  if (i != 0)
278                    x = 1; // dor nao inicial
279                  else copfon = 1; // dor inicial
280                else copfon = 1;  // nao e dor
281              if (x == 1)
282                if (fonaux[i+3] == 'i')
283                  if (fonaux[i+4] == 's') // dores
284                    if (fonaux[i+5] != ' ')
285                      x = 0;  // nao e dores
286                    else;
287                  else x = 0;
288                else
289                  if (fonaux[i+3] == 'a')
290                    if (fonaux[i+4] != ' ')
291                      if (fonaux[i+4] != 's')
292                        x = 0;
293                      else
294                        if (fonaux[i+5] != ' ')
295                          x = 0;
296                        else;
297                    else;
298                  else x = 0;
299              else x = 0;
300              if (x == 1)
301              { fonwrk[j] = 'D';
302                fonwrk[j+1] = 'o';
303                fonwrk[j+2] = 'R';
304                i = i + 5;
305              }// if
306              else copfon = 1;
```

```
307                 break;
308
309                 case 'F':  //se o caracter for F
310
311                   //F nao eh modificado
312                   copmud = 1;
313                 break;
314
315                 case 'G':  //se o caracter for G
316
317                   //gui -> gi
318                   if (fonaux[i+1] == 'o')
319                     if (fonaux[i+2] == 'i')
320                     { fonwrk[j] = 'G';
321                       fonwrk[j+1] = 'i';
322                       j += 2;
323                       i +=2;
324                     }//if
325                     //diferente de gui copia como consoante muda
326                     else copmud = 1;
327                   else
328
329                     //gl
330                     if (fonaux[i+1] == 'L')
331                       if (fonaux[i+2] == 'i')
332
333                         //gli + vogal -> li + vogal
334                         if ((fonaux[i+3]=='a')||
335                             (fonaux[i+3]=='i')||
336                             (fonaux[i+3]=='o'))
337                         { fonwrk[j] = fonaux[i+1];
338                           fonwrk[j+1] = fonaux[i+2];
339                           j += 2;
340                           i += 2;
341                         }//if
342                         else
343
344                           //glin -> lin
345                           if(fonaux[i+3] == 'N')
346                           { fonwrk[j] = fonaux[i+1];
347                             fonwrk[j+1] = fonaux[i+2];
348                             j += 2;
349                             i += 2;
350                           }/*if*/
351                           else copmud = 1;
352                       else copmud = 1;
353                     else
354
355                       //gn + vogal -> ni + vogal
356                       if (fonaux[i+1] == 'N')
357                         if((fonaux[i+2]!='a')&&
358                            (fonaux[i+2]!='i')&&
359                            (fonaux[i+2]!='o'))
360                           copmud = 1;
361                         else
362                         { fonwrk[j] = 'N';
363                           fonwrk[j+1] = 'i';
```

```
364                      j += 2;
365                      i++;
366                    }//else
367                  else
368
369                    //    ghi -> gi
370                    if (fonaux[i+1] == 'H')
371                      if (fonaux[i+2] == 'i')
372                      { fonwrk[j] = 'G';
373                        fonwrk[j+1] = 'i';
374                        j += 2;
375                        i +=2;
376                      }// if
377                      else copmud = 1;
378                    else copmud = 1;
379             break;
380
381             case 'H':  //se o caracter for H
382
383             //H eh desconsiderado
384             break;
385
386             case 'i':  //se o caracter for i
387
388               if (fonaux[i+2] == ' ')
389
390                 //is ou iz final perde a consoante
391                 if (fonaux[i+1] == 's')
392                 { fonwrk[j] = 'i';
393                   break;
394                 }// if
395                 else
396                   if (fonaux[i+1] == 'Z')
397                   { fonwrk[j] = 'i';
398                     break;
399                   }// if
400
401                 //ix
402                 if (fonaux[i+1] != 'X')
403                   copfon = 1;
404                 else
405                   if (i != 0)
406                     copfon = 1;
407                   else
408
409                     //ix vogal no inicio torna-se iz
410                     if ((fonaux[i+2]=='a')||
411                         (fonaux[i+2]=='i')||
412                         (fonaux[i+2]=='o'))
413                     { fonwrk[j] = 'i';
414                       fonwrk[j+1] = 'Z';
415                       j += 2;
416                       i++;
417                       break;
418                     }// if
419                     else
420
```

```
421                 //ix consoante no inicio torna-se is
422                 if (fonaux[i+2]=='C' || fonaux[i+2]=='s'
                       ) {
423                   fonwrk[j] = 'i';
424                   j++;
425                   i++;
426                   break;
427                 }// if
428                 else
429                   { fonwrk[j] = 'i';
430                     fonwrk[j+1] = 's';
431                     j += 2;
432                     i++;
433                     break;
434                   }// else
435         break;
436 
437         case 'J':  //se o caracter for J
438 
439           //J -> Gi
440           fonwrk[j] = 'G';
441           fonwrk[j+1] = 'i';
442           j += 2;
443         break;
444 
445         case 'K':  //se o caracter for K
446           //KT -> T
447           if (fonaux[i+1] != 'T')
448             copmud = 1;
449           break;
450 
451         case 'L':  //se o caracter for L
452 
453           //L + vogal nao eh modificado
454           if ((fonaux[i+1] == 'a')||
455               (fonaux[i+1] == 'i')||
456               (fonaux[i+1] == 'o'))
457             copfon = 1;
458           else
459 
460             //L + consoante -> U + consoante
461             if (fonaux[i+1] != 'H')
462               { fonwrk[j] = 'o';
463                 j++;
464                 break;
465               }// if
466 
467               //LH + consoante nao eh modificado
468               else
469                 if (fonaux[i+2] != 'a' &&
470                     fonaux[i+2] != 'i' &&
471                     fonaux[i+2] != 'o')
472                   copfon = 1;
473                 else
474 
475                   //LH + vogal -> LI + vogal
476                   { fonwrk[j] = 'L';
```

```
477                              fonwrk[j+1] = 'i';
478                              j += 2;
479                              i++;
480                              break;
481                            }
482                 break;
483 
484                 case 'M':  //se o caracter for M
485 
486                   //M + consoante -> N + consoante
487                   //M final -> N
488                   if ((fonaux[i+1] != 'a' &&
489                         fonaux[i+1] != 'i' &&
490                         fonaux[i+1] != 'o') ||
491                        (fonaux[i+1] == ' '))
492                   { fonwrk[j] = 'N';
493                     j++;
494                   }// if
495 
496                   //M nao eh alterado
497                   else copfon = 1;
498                 break;
499 
500                 case 'N':  //se o caracter for N
501 
502                   //NGT -> NT
503                   if ((fonaux[i+1] == 'G') &&
504                       (fonaux[i+2] == 'T'))
505                   { fonaux[i+1] = 'N';
506                     copfon = 1;
507                   }// if
508                   else
509 
510                     //NH + consoante nao eh modificado
511                     if (fonaux[i+1] == 'H')
512                       if ((fonaux[i+2] != 'a')&&
513                           (fonaux[i+2] != 'i')&&
514                           (fonaux[i+2] != 'o'))
515                         copfon = 1;
516 
517                       //NH + vogal -> Ni + vogal
518                       else
519                       { fonwrk[j] = 'N';
520                         fonwrk[j+1] = 'i';
521                         j += 2;
522                         i++;
523                       }
524                     else copfon = 1;
525                 break;
526 
527                 case 'o':  //se o caracter for o
528 
529                   //oS final -> o
530                   //oZ final -> o
531                   if ((fonaux[i+1] == 's') ||
532                       (fonaux[i+1] == 'Z'))
533                     if (fonaux[i+2] == ' ')
```

```
534                 { fonwrk[j] = 'o';
535                   break;
536                 }//if
537                 else copfon = 1;
538               else copfon = 1;
539             break;
540
541             case 'P':  //se o caracter for P
542
543               //PH -> F
544               if (fonaux[i+1] == 'H')
545               { fonwrk[j] = 'F';
546                 i++;
547                 newmud = 1;
548               }// if
549               else
550                 copmud = 1;
551             break;
552
553             case 'Q':  //se o caracter for Q
554
555               //Koi -> Ki (QUE, QUI -> KE, KI)
556               if (fonaux[i+1] == 'o')
557                 if (fonaux[i+2] == 'i')
558                 { fonwrk[j] = 'K';
559                   j++;
560                   i++;
561                   break;
562                 }// if
563
564               //QoA -> KoA (QUA -> KUA)
565               fonwrk[j] = 'K';
566               j++;
567             break;
568
569             case 'R':  //se o caracter for R
570
571               //R nao eh modificado
572               copfon = 1;
573             break;
574
575             case 's':  //se o caracter for s
576
577               //s final eh ignorado
578               if (fonaux[i+1] == ' ')
579                 break;
580
581               //s inicial + vogal nao eh modificado
582               if ((fonaux[i+1]=='a')||
583                   (fonaux[i+1]=='i')||
584                   (fonaux[i+1]=='o'))
585                 if (i == 0)
586                 { copfon = 1;
587                   break;
588                 }// if
589                 else
590
```

```
591                     //s entre duas vogais -> z
592                     if ((fonaux[i-1] != 'a')&&
593                         (fonaux[i-1]!='i')&&
594                         (fonaux[i-1]!='o'))
595                     { copfon = 1;
596                       break;
597                     }// if
598                     else
599 
600                       //SoL nao eh modificado
601                       if ((fonaux[i+1] == 'o') &&
602                           (fonaux[i+2] == 'L') &&
603                           (fonaux[i+3] == ' '))
604                       { copfon = 1;
605                         break;
606                       }// if
607 
608                       else
609                       { fonwrk[j] = 'Z';
610                         j++;
611                         break;
612                       }// else
613 
614                 //ss -> s
615                 if (fonaux[i+1] == 's')
616                   if (fonaux[i+2] != ' ')
617                   { copfon = 1;
618                     i++;
619                     break;
620                   }// if
621                   else
622                   { fonaux[i+1] = ' ';
623                     break;
624                   }// else
625 
626                 //s inicial seguido de consoante fica
                       precedido de i
627                 //se nao for sci, sh ou sch nao seguido de
                       vogal
628                 if (i == 0)
629                   if (!((fonaux[i+1] == 'C') &&
630                        (fonaux[i+2] == 'i')))
631                     if (fonaux[i+1] != 'H')
632                       if (!((fonaux[i+1] == 'C') &&
633                            (fonaux[i+2] == 'H') &&
634                            ((fonaux[i+3] != 'a')&&
635                             (fonaux[i+3]!='i')&&
636                             (fonaux[i+3]!='o'))))
637                       { fonwrk[j] = 'i';
638                         j++;
639                         copfon = 1;
640                         break;
641                       }// if
642 
643                 //sH -> X;
644                 if (fonaux[i+1] == 'H')
645                 { fonwrk[j] = 'X';
```

```
646                         i++;
647                         newmud = 1;
648                         break;
649                       }// if
650                       if (fonaux[i+1] != 'C')
651                       { copfon = 1;
652                         break;
653                       }// if
654
655                       //    sCh nao seguido de i torna-se X
656                       if (fonaux[i+2] == 'H')
657                         { fonwrk[j] = 'X';
658                           i += 2;
659                           newmud = 1;
660                           break;
661                         }// if
662                       if (fonaux[i+2] != 'i')
663                       { copfon = 1;
664                         break;
665                       }// if
666
667                       //sCi final -> Xi
668                       if (fonaux[i+3] == ' ')
669                       { fonwrk[j] = 'X';
670                         fonwrk[j+1] = 'i';
671                         i = i + 3;
672                         break;
673                       }// if
674
675                       //sCi vogal -> X
676                       if ((fonaux[i+3]=='a')||
677                           (fonaux[i+3]=='i')||
678                           (fonaux[i+3]=='o') )
679                       { fonwrk[j] = 'X';
680                         j++;
681                         i += 2;
682                         break;
683                       }// if
684
685                       //sCi consoante -> si
686                       fonwrk[j] = 's';
687                       fonwrk[j+1] = 'i';
688                       j += 2;
689                       i += 2;
690                     break;
691
692                     case 'T':  //se o caracter for T
693
694                       //TS -> S
695                       if (fonaux[i+1] == 's')
696                         break;
697
698                       //TZ -> Z
699                       else
700                         if (fonaux[i+1] == 'Z')
701                           break;
702                         else copmud = 1;
```

```
703                     break;
704
705           case 'V':  //se o caracter for V
706           case 'W':  //ou se o caracter for W
707
708             //V,W inicial + vogal -> o + vogal (U + vogal)
709             if (fonaux[i+1] == 'a'||
710                 fonaux[i+1] == 'i'||
711                 fonaux[i+1] == 'o')
712               if (i == 0)
713               { fonwrk[j] = 'o';
714                 j++;
715               }// if
716
717               //V,W NAO inicial + vogal -> V + vogal
718               else
719               { fonwrk[j] = 'V';
720                 newmud = 1;
721               }// else
722
723             else
724             { fonwrk[j] = 'V';
725               newmud = 1;
726             }// else
727           break;
728
729           case 'X':  //se o caracter for X
730
731             //caracter nao eh modificado
732             copmud = 1;
733           break;
734
735           case 'Y':  //se o caracter for Y
736           //Y jah foi tratado acima
737           break;
738
739           case 'Z':  //se o caracter for Z
740
741             //Z final eh eliminado
742             if (fonaux[i+1] == ' ')
743               break;
744
745             //Z + vogal nao eh modificado
746             else
747               if ((fonaux[i+1] == 'a')||
748                   (fonaux[i+1] == 'i')||
749                   (fonaux[i+1] == 'o'))
750                 copfon = 1;
751
752               //Z + consoante -> S + consoante
753               else
754               { fonwrk[j] = 's';
755                 j++;
756               }// else
757               break;
758
```

```
759                     default: //se o caracter nao for um dos jah
                           relacionados
760
761                       //o caracter nao eh modificado
762                       fonwrk[j] = fonaux[i];
763                       j++;
764                     break;
765                   }//switch
766
767                   //copia caracter corrente
768                   if (copfon == 1)
769                   { fonwrk[j] = fonaux[i];
770                     j++;
771                   }//if
772
773                   //insercao de i apos consoante muda
774                   if (copmud == 1)
775                     fonwrk[j] = fonaux[i];
776                   if (copmud == 1 || newmud == 1)
777                   { j++;
778                     k = 0;
779                     while (k == 0)
780                       if (fonaux[i+1] == ' ')
781                       //e final mudo
782                       { fonwrk[j] = 'i';
783                         k = 1;
784                       }//if
785                       else
786                         if ((fonaux[i+1]=='a')||
787                             (fonaux[i+1]=='i')||
788                             (fonaux[i+1]=='o'))
789                           k = 1;
790                         else
791                           if (fonwrk[j-1] == 'X')
792                           { fonwrk[j] = 'i';
793                             j++;
794                             k = 1;
795                           }//if
796                           else
797                             if (fonaux[i+1] == 'R')
798                               k = 1;
799                             else
800                               if (fonaux[i+1] == 'L')
801                                 k = 1;
802                               else
803                                 if (fonaux[i+1] != 'H')
804                                 { fonwrk[j] = 'i';
805                                   j++;
806                                   k = 1;
807                                 }//if
808                                 else i++;
809                       }
810
811             }//for
812           }//if
813         }//else
814
```

```
815         for (i = 0; i < s.length() + 3; i++)
816         //percorre toda a palavra, letra a letra
817
818           //i -> I
819           if (fonwrk[i] == 'i')
820             fonwrk[i] = 'I';
821           else
822
823             //a -> A
824             if (fonwrk[i] == 'a')
825             fonwrk[i] = 'A';
826             else
827
828              //o -> U
829              if (fonwrk[i] == 'o')
830               fonwrk[i] = 'U';
831               else
832
833                //s -> S
834                if (fonwrk[i] == 's')
835                 fonwrk[i] = 'S';
836                 else
837
838                  //E -> b
839                  if (fonwrk[i] == 'E')
840                   fonwrk[i] = ' ';
841                   else
842
843                    //Y -> _
844                    if (fonwrk[i] == 'Y')
845                     fonwrk[i] = '_';
846
847         //retorna a palavra, modificada, ao vetor que contem o texto
848         //System.out.println(new String(fonwrk));
849         component.remove(desloc);
850         component.add(desloc, new String(fonwrk));
851         desloc++;
852         j = 0; //zera o contador
853       }//for
854
855       str = u.listToStr(component);
856       //remonta as palavras armazenadas no vetor em um unico string
857
858       str = u.removeMultiple(str);
859       //remove os caracteres duplicados
860
861       return str.toUpperCase().trim();
862     }
863
864 }
```

UtilitiesMod.java

Código 10 – UtilitiesMod.java

```
1 package com.sun.mdm.index.phonetic.impl;
2
3 import java.util.concurrent.CopyOnWriteArrayList;
4
5 public final class UtilitiesMod {
6
7     public UtilitiesMod() {
8
9     }
10
11    public String removeMultiple (String str) {
12
13        //Remove text characters that are multiplied:
14        // ss -> s, sss -> s, rr -> r
15        char[] foncmp = new char[256];
16        //character array that stores the text without duplicates
17        char[] fonaux = new char[256];
18        //character array that stores the original text
19        char[] tip = new char[1];
20        //stores the previous character
21        int i, j;
22        j = 0;
23        tip[0] = ' ';
24        fonaux = str.toCharArray();
25        //the character array receives the original string
26
27        for (i = 0; i < str.length(); i++) {
28            //peruse the text, character by character
29            //runs through the text character by character
30            //is not number, space or S
31            if ((fonaux[i] != tip[0]) || (fonaux[i] == ' ')
32                ||((fonaux[i]>='0') && (fonaux[i]<='9'))
33                ||((fonaux[i]=='S')&&(fonaux[i-1]=='S')&&
34                ( (i>1) && (fonaux[i-2]!='S')))) {
35                foncmp[j] = fonaux[i];
36                j++;
37            }
38
39            tip[0] = fonaux[i];
40
41        }
42        //the string gets the text without duplicates
43        str = new String(foncmp);
44        return str.trim();
45    }
46
47    public String removeAccentuation (String str) {
48
49        //Replacing the accented characters by na accented
              characters
50
51        char aux[] = new char[256];
52        //matriz de caracteres onde o texto eh manipulado
53
54        int i;
55        aux = str.toCharArray();
56        //matriz recebe o texto
```

```
57
58          for (i = 0; i < str.length(); i++) {
59
60              switch (aux[i]) {
61                  case 'E':
62                      aux[i]='E';   //É -> E
63                      break;
64                  case 'Ê':
65                      aux[i]='E';   //Ê -> E
66                      break;
67                  case 'Ë':
68                      aux[i]='E';   //Ë -> E
69                      break;
70                  case 'Á':
71                      aux[i]='A';   //Á -> A
72                      break;
73                  case 'À':
74                      aux[i]='A';   //À -> A
75                      break;
76                  case 'Â':
77                      aux[i]='A';   //Â -> A
78                      break;
79                  case 'Ã':
80                      aux[i]='A';   //Ã -> A
81                      break;
82                  case 'Ä':
83                      aux[i]='A';   //Ä -> A
84                      break;
85                  case 'Ç':
86                      aux[i]='C';   //Ç -> C
87                      break;
88                  case 'Í':
89                      aux[i]='I';   //Í -> I
90                      break;
91                  case 'Ó':
92                      aux[i]='O';   //Ó -> 0
93                      break;
94                  case 'Õ':
95                      aux[i]='O';   //Õ -> 0
96                      break;
97                  case 'Ô':
98                      aux[i]='O';   //Ô -> 0
99                      break;
100                 case 'Ö':
101                     aux[i]='O';   //Ö -> 0
102                     break;
103                 case 'Ú':
104                     aux[i]='U';   //Ú -> U
105                     break;
106                 case 'Ü':
107                     aux[i]='U';   //Ü -> U
108                     break;
109                 case 'Ñ':
110                     aux[i]='N';   //Ñ -> N
111                     break;
```

```
112                 }
113             }
114             str = (new String(aux)).trim();
115             return str;
116         }
117
118         public String removeStrange (String str) {
119             //Elimina os caracteres que NAO sejam alfanumericos ou
                   espacos
120
121             char[] foncmp = new char[256];
122             //matriz de caracteres que armazena o texto original
123
124             char[] fonaux = new char[256];
125             //matriz de caracteres que armazena o texto modificado
126
127             int i, j,  //contadores
128                 first; //indica se exitem espacos em branco antes do
                       primeiro
129                     //caracter: se 1 -> existem, se 0 -> nao
                           existem
130
131             j = 0;
132             first = 1;
133             fonaux = str.toCharArray();
134             //matriz de caracteres recebe o texto
135
136             for (i = 0; i < 256; i++)
137                 foncmp[i] = ' ';
138
139             //branqueia a matriz de caracteres
140
141             for (i = 0; i < str.length(); i++) {
142                 //percorre o texto, caracter a caracter
143
144                 //elimina os caracteres que nao forem alfanumericos ou
                       espacos
145                 if (((fonaux[i]>='A')&&
146                     (fonaux[i]<='Z')) ||
147                     ((fonaux[i]>='a')&&
148                     (fonaux[i]<='z')) ||
149                     ((fonaux[i]>='0')&&
150                     (fonaux[i]<='9')) ||
151                     (fonaux[i] == '&')   ||
152                     (fonaux[i] == '%')   ||
153                     (fonaux[i] == '_')   ||
154                     ((fonaux[i] == ' ') && first == 0)) {
155                         foncmp[j] = fonaux[i];
156                         j++;
157                         first = 0;
158                 }// if
159             }//for
160             str = new String(foncmp);
161             return str.trim();
162         }
163
164         public CopyOnWriteArrayList<String> strToList(String str) {
```

```
165
166         //armazena o texto de um string em um vetor onde
167         //cada palavra do texto ocupa uma posicao do vetor
168
169         str = str.trim();
170
171         char[] fonaux = new char[256];
172         //matriz de caracteres que armazena o texto completo
173
174         char[] foncmp = new char[256];
175         //matriz de caracteres que armazena cada palavra
176
177         CopyOnWriteArrayList<String> component = new
               CopyOnWriteArrayList<String>();
178         //vetor que armazena o texto
179
180         String aux = new String();
181
182         int i, j,   //contadores
183             pos,    //posicao da matriz
184             rep,    //indica se eh espaco em branco repetido
185             first; //indica se eh o primeiro caracter
186
187         first = 1;
188         pos = 0;
189         rep = 0;
190
191         fonaux = str.toCharArray();
192         //matriz de caracteres recebe o texto
193
194         for (j = 0; j < 256; j++)
195         foncmp[j] = ' ';
196         //branqueia matriz de caracteres
197
198         for (i = 0; i < str.length(); i++) {
199             //percorre o texto, caracter a caracter
200
201             //se encontrar um espaco e nao for o primeiro caracter,
202             //armazena a palavra no vetor
203             if ((fonaux[i] == ' ') && (first != 1)) {
204                 if (rep == 0) {
205                     component.add((new String(foncmp)).trim());
206                     pos = 0;
207                     rep = 1;
208                     for (j = 0; j < 256; j++)
209                         foncmp[j] = ' ';
210                 }
211             }
212
213             //forma a palavra, letra a letra, antes de envia-la a
                   uma
214             //posicao do vetor
215             else {
216                 foncmp[pos] = fonaux[i];
217                 first = 0;
218                 pos++;
219                 rep = 0;
```

```
220                 }
221             }
222             if (foncmp[0] != ' ')
223                 component.add((new String(foncmp)).trim());
224             return component;
225 
226         }
227 
228         public String listToStr(CopyOnWriteArrayList<String> vtr) {
229 
230             //converte o texto armazenado em um vetor para um
                    unico string
231 
232             char[] foncmp = new char[256];
233             //matriz de caracteres que armazena o texto completo
234             char[] auxChar = new char[256];
235             //matriz de caracteres que armazena cada palavra
236 
237             String auxStr;
238             String str;
239             int i, j, desloc;
240 
241             desloc = 0;  //deslocamento dentro da matriz
242 
243             for (i = 0; i < 256; i ++)
244                 foncmp[i] = ' ';
245                 //branqueia a matriz de caracteres
246 
247             for (j = 0; j < vtr.size(); j++) {
248                 //percorre o vetor, palavra a palavra
249                 auxStr = (vtr.get(j)).toString().trim();
250                 auxChar = auxStr.toCharArray();
251                 for (i = 0; i < auxStr.length(); i++)
252                     foncmp[desloc + i] = auxChar[i];
253                 desloc = desloc + auxStr.length() + 1;
254             }
255 
256             str = new String(foncmp);
257             //string recebe o texto completo
258 
259             return str.trim();
260 
261         }
262 
263 }
```