GUSTAVO DE GEUS

EXECUÇÃO DE EXPERIMENTOS COM FUNÇÕES DE SIMILARIDADE EM UM AMBIENTE PARALELO

FLORIANÓPOLIS
Dezembro 2012

GUSTAVO DE GEUS

EXECUÇÃO DE EXPERIMENTOS COM FUNÇÕES DE SIMILARIDADE EM UM AMBIENTE PARALELO

Trabalho de conclusão de curso apresentado pelo acadêmico Gustavo de Geus à banca examinadora do Curso de Graduação em Sistemas de Informação da Universidade Federal de Santa Catarina como requisito parcial para obtenção do título de Bacharel em Sistemas de Informação.

Orientadora:
Profª. Drª. Carina Friedrich Dorneles

UNIVERSIDADE FEDERAL DE SANTA CATARINA
CENTRO TECNOLÓGICO
DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
SISTEMAS DE INFORMAÇÃO

FLORIANÓPOLIS
Dezembro 2012

GUSTAVO DE GEUS

EXECUÇÃO DE EXPERIMENTOS COM FUNÇÕES DE SIMILARIDADE EM UM AMBIENTE PARALELO

Orientadora:

_____

Profª. Drª. Carina Friedrich Dorneles

Banca Examinadora:

_____

Prof. Dr. Mario Dantas

_____

Prof. Dr. Ronaldo dos Santos Mello

FLORIANÓPOLIS
Dezembro 2012

AGRADECIMENTOS

Agradeço principalmente a Deus por mais essa conquista em minha vida. A minha esposa Sarah, pela enorme paciência e apoio nos momentos mais difíceis. Agradeço aos meus pais por tudo que fizeram por mim durante toda a vida. Agradeço também à professora Carina pela orientação durante todo o trabalho mesmo nas situações mais adversas.

RESUMO

Com o volume e falta de padronização de dados crescendo a cada dia, o uso de funções de similaridade torna-se cada vez mais importante para quem deseja gerenciar esse grande volume da dados. Essas funções podem ser aplicadas em diversos tipos de dados, desde bancos de dados relacionais, XML ou até mesmo bases multimídia. Porém, a execução destas funções tem um custo computacional bastante elevado, o que inviabiliza a sua aplicação em um ambiente contendo um grande volume de dados, e alta escalabilidade.

Uma das propostas para tornar possível a execução dessas funções de similaridade em grandes volumes de dados é a utilização de programação paralela em conjunto com técnicas de blocagem. Dessa forma, a programação paralela seria implementada focando o problema de escalabilidade, tendo em vista que seu conceito é baseado em uma *thread* pai que executa um procedimento chamado de *thread-join*, o qual cria algumas *threads* filhas e, enquanto essas *threads* filhas são executadas em paralelo, a *thread* pai aguarda a conclusão de todas as tarefas, fazendo com que um processo possa ser distribuído em diferentes nós, a fim de melhorar o seu desempenho. O agrupamento dos dados em blocos indexados por uma chave permite que seja reduzida a quantidade de comparações com dados que não possuem relação nenhuma com o que é desejado, viabilizando a analise de um grande volume de dados.

Tendo como base o cenário descrito acima, a proposta desse trabalho é criar uma arquitetura utilizando alguns conceitos de programação paralela e blocagem de dados para que ao final seja possível realizar experimentos, aplicando funções de similaridade utilizando dados blocados ou não no ambiente preparado. A implementação utiliza uma estratégia mais simples em termos de desenvolvimento, porém mantendo o ganho em desempenho proporcionado pela programação paralela. Dessa forma, pretende-se chegar a resultados que demonstrem as diferenças entre a utilização ou não desse modelo de programação para resolver o problema de desempenho na execução das funções.

Palavras-chave: Programação paralela, funções de similaridade, blocagem de dados.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1. Introdução

Um dos maiores desafios para os profissionais da área de sistemas de informação e banco de dados é gerenciar o grande volume de dados que vem sendo gerado a cada dia. Para piorar o cenário, esses dados são gerados de maneira não estruturada por diferentes usuários e softwares, fazendo com que surjam inúmeras replicações de dados, ou seja, um mesmo objeto do mundo real representado de maneiras diferentes no mundo digital. Nesse contexto, o uso do operador de igualdade na manipulação desses dados não é adequado e, devido a isso, surgem as funções de similaridade para auxiliar no controle desses dados. Hoje já existem diversas funções de similaridade disponíveis, tanto para uso em valores atômicos [Cohen et al. 2003a, Chapman 2004, Cohen et al. 2003b, Tejada et al. 2001, Navarro 2001, de Lima 2002] quanto para aplicação em conjuntos de dados. As funções para valores atômicos podem ser divididas em baseada em caracter (*Levenshtein(), Jaro())* ou *token* (*SoftTFIDF(), MongeElkan()* e *Jaccard())*, as funções para uso em conjunto de dados dividem-se em: expressões algébricas [Guha et al. 2004, Schallehn et al. 2004, Chaudhuri et al. 2003, Yu et al. 2003, Motro 1988]; e algoritmos [Zhao and Ram 2005, Shen et al. 2005, Michalowski et al. 2005, Bilenko et al. 2003, Bilenko and Mooney 2003, Doan et al. 2003, Tejada et al. 2001, Huffman and Steier 1995]. Essas funções são muito usadas no processo de integração de dados, onde se deve identificar um mesmo objeto no mundo real representado de forma diferente em fontes de dados diversas. Também são utilizados no processo de *Data Cleaning* e até mesmo em consultas a dados multimídia como imagens e arquivos de sons. O problema da deduplicação, como é chamado o processo de identificar e tratar dados duplicados é algo que tem a sua complexidade aumentada à medida que o tamanho das bases de dados também aumentam.

A utilização dessas funções aplicadas em um grande volume de dados complexos é algo bastante custoso computacionalmente, fazendo com que sua execução se torne um processo bastante demorado e consequentemente caro. Além do custo normal de se ter um equipamento exclusivo para que o processamento seja realizado, o que pode durar alguns dias, entram ainda custos com mão de obra para monitoramento e controle da execução das funções de similaridade, além de se ter o risco de ocorrer algum problema no decorrer da transação.

Essas dificuldades incentivaram estudos em dois sentidos para a solução do problema. Uma das propostas é construir blocos de dados e a outra é fazer a execução do processo de forma paralela. Foram feitos alguns estudos juntando os dois conceitos (*P-Swoosh* [KAWAI et. al. 2006] e *P-Canopy* [DAL BIANCO et. al. 2009]), conseguindo-se blocar dados de maneira mais otimizada. Para a execução das funções de similaridade utilizando programação paralela não foi encontrado nenhum trabalho especifico ou que tivesse uma abordagem semelhante à apresentada nesse trabalho.

A programação paralela visa permitir que o processo seja distribuído em diferentes processos filhos, ao invés de ser disparado um único processo. Dessa forma, é possível delegar a cada núcleo de um processador a execução de uma parte do processo pai, utilizando todo o recurso disponível dividindo o trabalho entre eles. Além da economia de tempo, esse modelo de programação viabiliza a

execução de processos extremamente onerosos computacionalmente em ambientes mais simples, reduzindo até mesmo o custo de um projeto[1]. A idéia principal desse trabalho é explorar a possibilidade de execução das funções de similaridades utilizando programação *multi-thread* de maneira simples, porém eficaz. Para isso foram realizados diversos experimentos, com funções de similaridade diferentes, tipos de dados diferentes e utilizando ou não técnicas de blocagem a fim de se obter resultados que possam realmente demonstrar qual a real diferença entre a adoção do processamento paralelo das funções de similaridade.

O trabalho é organizado da seguinte forma: no Capítulo 3, fundamentação teórica, são explicados, com base na bibliografia, os conceitos dos temas principais utilizados para esse trabalho, sendo eles as funções de similaridade, técnicas de blocagem e paralelização de processos. O capitulo 4 apresenta alguns trabalhos que abordam assuntos semelhantes ao estudado no presente trabalho e uma comparação entre os três trabalhos relacionados e o trabalho apresentado. O Capítulo 5 foca no P-BSim (*Parallel Blocking Similarity*) que foi desenvolvido nesse trabalho. As seções do Capítulo 5 apresentam uma visão geral do que é o P-BSim, os conceitos de blocagem e processamento paralelo de funções utilizados além da implementação, onde são mostrados os artefatos produzidos para compor toda a arquitetura utilizada neste trabalho assim como a tecnologia utilizada em cada um. Ao fim da seção os resultados obtidos são apresentados de forma comparativa, demonstrando os ganhos quando utilizado os conceitos desenvolvidos no P-BSim. Por fim, o Capítulo 6 apresenta as conclusões alcançadas através do trabalho e quais os próximos passos a serem realizados.

[1] Parallel computing - Wikipedia http://en.wikipedia.org/wiki/Parallel_computing

2. Fundamentação teórica

A execução de funções de similaridade tem um custo computacional bastante elevado, o que inviabiliza a sua aplicação em um ambiente contendo um grande volume de dados. Alguns estudos, como [SANTOS, W et. al. 2007] e [DOS SANTOS, W 2008] têm sido realizados no sentido de tornar possível a execução dessas funções de similaridade em grandes volumes de dados, com a utilização de programação paralela em dados previamente blocados, afim de diminuir a quantidade de comparações desnecessárias.

2.1. Funções de similaridade

Funções de similaridade são aplicadas em situações onde o uso do operador de igualdade não é adequado pois os valores comparados apresentam diferentes representações, mesmo quando se referem ao mesmo objeto do mundo real, por exemplo, quando ocorre duplicação de dados como nomes próprios, datas, endereços, nomes de cidades e países, entre outros. Esta questão tem motivado pesquisa nas mais diversas áreas e contextos, desde questões relacionadas à consulta sobre bases de dados com mais de uma representação do mesmo objeto até a integração de fontes que possuem dados referentes ao mesmo domínio.

Para cada tipo de dado existe uma função mais adequada [Dorneles, et al 2009]. Dessa forma é possível identificar quando duas instâncias representam a mesma entidade no mundo real. Sua aplicação se dá em documentos textuais, desde simples campos de registro até documentos inteiros. Porém, já foram desenvolvidas técnicas de comparação entre dados binários, documentos XML. Basicamente, uma função de similaridade calcula um escore para um par de valores, onde esse escore varia entre 0 e 1. Quanto maior o valor do escore, mais semelhantes são os valores, onde 1 os valores são totalmente igual e 0 totalmente diferentes.

Para definir se dois valores de dados se referem ao mesmo objeto do mundo real, o resultado das funções deve trabalhar juntamente com um valor de limiar previamente definido. Esse valor deve ser escolhido com muita cautela, pois é ele que vai definir se, dado um par de dados, eles representam o mesmo objeto ou não. Por exemplo, dado os valores “casa” e “caza”, com um valor de limiar igual a 0.8, ao aplicar uma função de similaridade qualquer sobre esse valor, o retorno será um escore. Caso o escore retornado seja um valor maior ou igual a 0.8, os dois objetos são definidos como tendo a mesma representação no mundo real, caso contrário, possuem representações diferentes. Para esse caso, onde o problema é um erro tipográfico, a execução deveria retornar que os objetos representam a mesma coisa no mundo real.

2.2. Blocagem de dados

As técnicas de blocagem têm como objetivo principal limitar o número de comparações realizadas no processo de deduplicação, tendo em vista que, no pior dos casos, o quadrático ou produto cartesiano (PC), são realizadas comparações com dados com pouca ou nenhuma relação. Fazendo uma analogia com sistemas de banco de dados, a blocagem funciona como um processo de indexação por similaridade.

Para realizar as blocagens, se utiliza um predicado de blocagem, o qual consiste em gerar uma espécie de chave através de campos pré-definidos, que é utilizada para classificar os dados em blocos. A definição desse predicado é uma tarefa de extrema importância para que se tenha sucesso em todo o processo. Ele deve sempre levar em conta a qualidade da informação contida no campo selecionado para fazer parte da chave.

A blocagem padrão, referenciada na literatura como *Standard Blocking [Christen 2007]* (SB), é uma técnica tradicional e simples, que agrupa os registros de acordo com uma chave de blocagem predefinida. Nesse modo de blocagem, o melhor caso fornece blocos com a mesma quantidade de registros e funciona da seguinte forma: para cada registro lido, gera-se um conjunto de chaves de blocagem; ao final, apenas registros que geraram pelo menos uma chave de blocagem em comum serão comparados e cada par de registros gerado é, então, chamado de par candidato.

O método *Bigram Indexing (BI) [Baxter et al. 2003, Bilenko et al. 2006]* permite uma abordagem difusa da blocagem (*fuzzy blocking*), possibilitando que alguns erros tipográficos possam ser captados. A ideia básica é converter a chave de blocagem em uma lista de bigramas (sub-strings de dois caracteres), por exemplo, uma chave de blocagem com o valor 'priscila' gera os bigramas ('pr','ri','is','sc','ci','il','la'). A partir desta lista de bigramas, sub-listas são geradas usando todas as permutações possíveis sob um limite t com valor entre zero e um. As listas resultantes são convertidas em chaves de blocagem e o processo de blocagem continua da mesma forma do que o SB. Utilizando o exemplo anterior, se aplicado o método já descrito com um limite de valor 0.8, o resultado consiste nas seguintes chaves geradas: *prisscciilla, prriisciilla, prriisscciil, prriisscila, prriisscilla, prrisccciilla, riisscciilla*. Dessa forma, todos os registros com a chave de blocagem 'priscila' serão inseridos em sete blocos diferentes, um para cada chave resultante.

A técnica *Sorted Neighbourhood (SN) [Hernandez and Stolfo 1998, Bilenko et al. 2006]* é baseada na ordenação dos registros por uma chave de blocagem e na utilização de uma janela de tamanho fixo que percorre todo o volume de dados. Dessa forma apenas os registros que estiverem na mesma janela são considerados pares candidatos.

A principal vantagem do *Sorted Neighbourhood* perante as outras técnicas apresentadas é que o número de comparações fica reduzido ao tamanho da janela selecionada e tende a ser menor. Porém, uma das desvantagens é que, caso algum registro tenha um erro em sua chave de blocagem, ele não é comparado com todos os semelhantes, como por exemplo para o nome *Kelli* e *Quelli*, um erro de grafia faria com que as duas entidades semelhantes não fossem comparadas.

A despeito do pior caso quadrático em termos de comparação, o que se observa é que a maior parte dos registros comparados possui pouca ou nenhuma relação, sendo a comparação desnecessária.

Mais ainda, o número de pares encontrados cresce linearmente com o tamanho da base [Baxter et al. 2003].

O predicado de blocagem [Hernandez and Stolfo 1998] é utilizado para definir quais os atributos dos registros e quais transformações são aplicadas nas técnicas de blocagem. Um predicado é uma disjunção de conjunções, onde cada termo da conjunção define uma função de transformação sobre o registro. Um exemplo de predicado é $P = (\text{nome} \wedge \text{ano\_de\_nascimento}) \vee (\text{sobrenome} \wedge \text{cidade})$. Quando aplicado a um registro, o predicado de blocagem é capaz de gerar uma chave de blocagem para cada conjunção.

Dessa forma, consegue-se acessar dados previamente "filtrados", evitando as comparações desnecessárias. Cada uma das chaves geradas através do predicado de blocagem contém um conjunto de valores, que são buscados cada vez que algum valor compartilhar essa chave. Deve-se ter cuidado na escolha dos atributos do predicado de blocagem, pois são eles que vão definir a qualidade das chaves geradas.

2.3. Paralelização de algoritmos

Com as mudanças nas arquiteturas computacionais, a cada ano são lançados processadores com mais núcleos. O ramo de desenvolvimento não pode ficar atrás. Tendo em vista que a maioria das linguagens de programação fornece suporte nativo para programação sequencial, esses núcleos ficam ociosos na execução de programas comuns.

Porém, para problemas de cunho cientifico ou até mesmo na área de *Business Inteligence (BI),* onde são realizados processamentos de dados extensivos e custosos, exigiu-se uma atenção maior no que se refere à exploração desses recursos computacionais. Com isso, começaram a ser estudadas formas de paralelizar a execução de algoritmos, fazendo com que cada um dos núcleos do processador ficasse com uma parte da tarefa. Dessa forma, o processamento como um todo seria realizado em um tempo teoricamente menor[2].

A programação paralela vem sendo aprimorada em diversas linguagens de programação. Muitas delas já abstraíram o controle de *threads*[3], deixando o desenvolvedor mais focado nas regras de negócio do problema do que no controle de *threads*. Uma das linguagens que tem um suporte à programação paralela bastante desenvolvida é o Scala[4], uma linguagem que trabalha sob o paradigma funcional ou orientado a objetos. Devido a ser uma linguagem com integração transparente com o JAVA, muitas empresas estão adotando o Scala quando se necessita de desempenho e escalabilidade. Um

[2] Programação paralela - http://pt.wikipedia.org/wiki/Computa%C3%A7%C3%A3o_paralela
[3] Linha ou Encadeamento de execução (em inglês: *Thread*), é uma forma de um processo dividir a si mesmo em duas ou mais tarefas que podem ser executadas concorrentemente. (Wikipedia)
[4] Scala (*Scalable language*) é uma linguagem de programação de propósito geral, diga-se multiparadigma, projetada para expressar padrões de programação comuns de uma forma concisa, elegante e *type-safe*. Ela incorpora recursos de linguagens orientadas a objetos e funcionais. Também é plenamente interoperável com Java.(www.scala-lang.org)

exemplo é o *Twitter*, que mudou toda a sua fila de mensagens que antes era feita em *Ruby* para Scala [Scala in the Enterprise, 2012].

Na área de banco de dados, a programação paralela também vem sendo estudada, tendo em vista que o volume de dados vem crescendo a cada dia mais e, analisar, gerenciar e manter esses dados tem sido uma tarefa bastante onerosa. No problema da deduplicação, problema tratado nesse trabalho, é possível encontrar diversas técnicas de paralelização de algoritmos vem sendo empregadas, que serão apresentadas mais adiante.

3. Trabalhos relacionados

Essa seção tem como objetivo apresentar alguns trabalhos já realizados, relacionados ao tema estudado nesse trabalho. Ao final, é feita uma análise comparativa entre eles a fim de deixar claro qual o foco principal e diferencial entre cada um deles e o presente trabalho.

3.1. P-Swoosh

O P-Swoosh [Kawai et. al. 2008] foca no problema de Resolução de Entidades (ER), e o objetivo é trabalhar em cima de duas funções: *match* e *merge*, ou combinar e fundir. A função match é encarregada de identificar dados duplicados, enquanto a merge mescla dados semelhantes a fim de inseri-los em um mesmo contexto. Esse tipo de aplicação (ER) vem sendo usado em problemas que vão desde o tratamento de dados para gestão de relacionamento com cliente até a mineração de dados para combater o terrorismo, problemas esses que muitas vezes determinam o modo de operação das funções *match* e *merge*.

Normalmente, a maioria dos trabalhos focados em ER são feitos apenas para um problema em especifico, porém o P-Swoosh trabalha com uma abordagem de um ER genérico, o qual deve ter a capacidade de operar sobre um maior número de problemas distintos. Ele também é capaz de operar sob um ambiente paralelo, tendo em vista que todos os processos envolvidos em um ER exigem recursos computacionais em grande quantidade. Essa necessidade fez com que fosse proposto um modelo de execução paralela, onde fosse possível dividir todas as combinações de registros possíveis para que a comparação dos registros pudesse ser atribuída a diferentes processadores. O algoritmo paralelo para ER genérico descrito, o P-Swoosh, foi experimentado sob uma base de dados do Yahoo! e demonstrou uma escalabilidade quase linear para um número de processadores que variou entre dois e quinze.

O procedimento básico para a resolução de entidades genéricas foi definido como R-Swoosh. O benefício dessa abordagem genérica é que fica possível avaliar precisão e desempenho de maneira independente, ou seja, é possível que o usuário defina qual a precisão de *merge* e *match* que ele deseja trabalhar. Dessa maneira, a intenção do trabalho é desenvolver um método otimizado para fornecer resultados completos para “funções de *merge*” e “combinações em uma caixa-preta“.

Na combinação baseada em paralelismo, é necessário definir o par de registros a ser comparado em um processador e alocar os registros nele. Para isso, foram definidas duas funções: a *responsible* que avalia o par de dados a partir de um processador, e caso esse processador for responsável por esse par de dados ela retorna verdadeiro; e a *scope* que retorna qual processador possui o registro pai, aquele que deu origem aos outros valores analisados. Para esse processo foram implementadas duas formas paralelas: o esquema de replicação completo e o esquema em rede.

Na replicação completa, todos os processadores possuem um cópia de todos os registros existentes e dessa forma consegue trabalham de maneira independente um do outro. O problema é que

dessa forma ocorre uma sobrecarga no armazenamento de dados, além de gerar trabalho redundante nos processadores.

Já o modelo em rede, os registros são divididos em grupos disjuntos e esses grupos são delegados a processadores diferentes. Dessa forma, o gargalo de armazenamento cai consideravelmente e, em questão de comunicação, a rede não precisa enviar um registro "fundido" para toda a rede, ela envia apenas para os processadores que possuem correlação com o dado.

O algoritmo do P-Swoosh trabalha com dois conjuntos de registros: R e R0, onde R contém os registros a serem pareados e R0 será o conjunto-resultado. O conjunto R0 é subdividido em nós escravos que ficam sobre responsabilidade de um nó mestre, o qual controla a execução das funções de similaridade de maneira distribuída. A sua execução ocorre da seguinte maneira: um registro de R é definido como sendo o alvo e o mesmo é removido de R. Após isso é feita uma comparação do registro alvo com todos os registros contidos em R0. Caso exista concordância entre eles, ou seja, o resultado da comparação entre os registros atingiu um score mínimo definido, todos os registros são mesclados, gerando um novo com campos multidimensionais e dessa forma os registros que o geraram são removidos. Esse novo registro é denominado de canônico. Caso não haja concordância, o registro alvo é inserido em R0 e esse processo é realizado até que o conjunto R seja esvaziado.

O algoritmo P-Swoosh trabalha com dois métodos de balanceamento de carga a fim de distribuir da melhor maneira o processo de comparação em diferentes nós. Para isso, ele explora tanto o "balanceamento de carga horizontal", o qual equilibra a carga entre todos os nós escravos da rede, quanto o "balanceamento de carga vertical", que visa equilibrar os trabalhos entre o mestre e os nós escravos.

Um fator que influencia diretamente nos resultados alcançados é a escolha do tamanho de janela de dados utilizados pelos algoritmos. Dessa forma, foram exploradas tanto a estratégia denominada de "Adaptação de registro base", a qual tenta prever o custo computacional entre mestre e nós escravos levando em conta apenas a quantidade de registros, quanto a estratégia de "adaptação assintótica", a qual tenta estimar o número de comparações entre nós mestres e escravos tendo como base o processo de comparação mais recente.

3.1.1. Experimentos

Os experimentos do algoritmo P-Swoosh foram realizados sob um subconjunto de dados sobre compras do Yahoo. O ambiente utilizado para os testes possuía uma base de dados com os registros de compras cadastrados no site Yahoo! Shopping com aproximadamente 8GB. Foram avaliados diferentes cenários e exploradas diferentes possibilidades, como o conhecimento e não conhecimento do domínio. Para cada um dos testes eram inseridos os parâmetros necessários para uma melhor desempenho. Quanto a recursos computacionais, todo o desenvolvimento foi feito em *JAVA* e foram executados em processadores *Pentium* de 2.8 GHz com 1GB de memória RAM. A avaliação do algoritmo *P-Swoosh* foi baseada nas seguintes métricas:

• **computação agregada**: consiste em calcular qual dos processos de comparação foi mais custoso. Assim foi possível se ter uma ideia bastante precisa da otimização do processo quando executado em paralelo.

• **esforço máximo**: mede a quantidade de comparações executadas por um nó escravo ou mestre. Esses valores, se somados, retornam o tempo total de execução.

• ***speedup***: mede a eficiência do algoritmo paralelo verificando a relação entre o aumento de velocidade com o número de comparações em um processo em série (*R-Swoosh*) dividido pelo esforço máximo em um processo paralelo (*P-Swoosh*).

• **comunicação**: mede o número total de registros trocados entre nó mestre e escravo.

Para o primeiro teste, o algoritmo foi executado em 500 mil registros assumindo que o domínio não era conhecido. Nesse cenário os resultados foram os seguintes: utilizando o conceito de *batch window*, foi possível reduzir a agregação computacional e reduzir bastante o custo de comunicação, porém ficou evidente que se for definida uma janela muito grande, os benefícios do paralelismo não são tão aproveitados. Quanto ao balanceamento de carga, na abordagem horizontal a escolha de um nó escravo com um número mínimo de registros é ligeiramente melhor do que uma estratégia que trabalhe sobre o conceito de *Round Robin*. Já para a abordagem vertical, o uso de uma janela adaptável é melhor que a de janela fixa. Quanto à comparação com outros algoritmos, o *P-Swoosh* apresentou um desempenho bem melhor sob um ambiente paralelo.

Em um segundo momento, os resultados foram obtidos em um cenário onde o domínio do problema era conhecido. Para isso, foram utilizados cinco mil registros fornecidos pelo *Yahoo! Shopping*, porém exclusivamente os que tinham alguma referência ao *iPod*.

A diferença entre esse experimento e o anterior, onde não se tinha domínio do problema, é que os registros iniciais são ordenados pelo atributo preço e todos os nós (mestre e escravo) usam os índices de preços para obter os R' candidatos a um registro de destino.

Nesse ambiente, o *P-Swoosh* também conseguiu obter um benefício do paralelismo, onde a janela adaptativa teve uma vantagem muito pequena sobre a janela fixa, já que na maioria dos casos todas as comparações eram feitas pelo nó mestre.

Para avaliar a escalabilidade do algoritmo, a mesma quantidade de dados foi dividida em subgrupos de 15, 10 e 5 mil registros. Normalmente, o custo evolui de forma quadrática em relação ao número de registros, porém na execução paralela isso teve uma melhora significativa, algo relacionado a 10% do valor obtido na execução dos algoritmos em sequencial. A escalabilidade também ficou bastante estável em todos os volumes de dados.

3.2. A Scalable Parallel Deduplication Algorithm

Bases de dados confiáveis e consistentes são algo primordial quando se fala em análise de tendência, detecção de fraudes, *business inteligence* (BI) e sistemas de apoio à decisão. Porém, essas bases de dados são bastante difíceis de obter, tendo em vista que no mundo real as bases de dados

normalmente possuem muitos dados inválidos ou até mesmo incompletos. Uma das coisas que invalida um dado é a sua duplicação, e esse trabalho tem como objetivo detectar a existência desse tipo de inconsistência em alguns cenários como bibliotecas digitais, registros de saúde, dados bancários, entre outros.

Para a realização do processo de deduplicação, é possível utilizar diversas abordagens, desde as mais simples que definem um conjunto de registros como identificados até as mais avançadas que trabalham com um conjunto de regras. O relacionamento probabilístico é um exemplo de abordagem onde existem atributos verdadeiros e falsos, os quais vão determinar se um atributo pertence a um determinado grupo estabelecido. Tendo em vista que essa estratégia pode ser dividida em determinar os registros a se comparar, realizar as comparações e verificar os resultados dessa comparação, pode-se afirmar que no pior caso o custo computacional será quadrático.

Alguns desafios devem ser superados a fim de que se possa alcançar um nível otimizado de escalabilidade, sendo eles o custo computacional, tendo em vista o grande volume de dados, a demanda de armazenamento, o que normalmente torna o ambiente distribuído, e por fim a variabilidade dos dados trabalhados. No trabalho [Santos, et. al. 2007], é apresentada uma paralelização escalável do algoritmo de relacionamento probabilístico em uma tentativa de superar os desafios citados.

Em casos onde não existe um identificador único ou não se sabe o tamanho do registro é necessário desenvolver algumas técnicas mais elaboradas nos processos de deduplicação, para reconhecer dados duplicados com a menor taxa de falha possível.

3.2.1. O processo de deduplicação

Em relação ao processo de deduplicação, ele pode ser dividido em sub-tarefas, começando pela limpeza e padronização da base de dados a ser avaliada, definição do número de comparações realizadas utilizando técnicas de blocagem, comparação dos registros e classificação dos resultados.

Na fase inicial, são eliminados dados totalmente desconexos, etapa que visa auxiliar a execução das próximas. Na segunda fase, são aplicadas técnicas de blocagem, onde um dado é classificado através de uma chave e esse é inserido em um grupo identificado por ela. Isso é uma preparação para a próxima fase, pois o intuito é diminuir o número de comparações, fazendo com que sejam comparados apenas dados do mesmo bloco. Esse processo é feito através de funções de similaridade, que avaliam o dado por diferentes abordagens dependendo da função escolhida para o procedimento. Por fim é executado o processo de classificação, onde os dados são ranqueados com base no escore obtido após a realização das comparações com os dados da base.

O trabalho utilizou dois algoritmos básicos, sendo eles:

- **Algoritmo sequencial**: no modo sequencial, o algoritmo trabalha da seguinte forma: cada registro lido do banco de dados gera sua chave de blocagem através do predicado já estabelecido. Se houver um bloco esse registro é inserido nele. Caso não exista, esse bloco é criado e o registro inserido.

- **Algoritmo paralelo**: nessa parte, é apresentada a sugestão de algoritmo paralelo assim como o ambiente utilizado na execução descrito a seguir.

3.2.2. Anthill

O ambiente *Anthill* viabiliza explorar o paralelismo de três formas: paralelismo de tarefas, paralelismo de dados e assíncronia. Dessa forma, fica viável executar os processos divididos em vários estágios de um *pipeline*, utilizando o paralelismo de tarefas, com várias réplicas de dados, utilizando dados em paralelo e, a assíncronia garante que a execução possa ser feita em diferentes pontos.

Esse ambiente fornece uma abstração chamada de fluxo rotulado (*labeled stream*), onde cada mensagem que esteja sendo executada no ambiente distribuído pode ser rotulada a um filtro específico definido em um mapeamento. Assim, é possível ter total controle sobre os encaminhamentos de mensagens da aplicação e assim ter um ambiente bastante equilibrado e controlado.

3.2.3. Estratégia de paralelização

A estratégia de deduplicação paralela proposta é baseada em quatro filtros: *ReaderComparator, Blocking, Classifier, e o Merger*. O *ReaderComparator* é encarregado tanto de definir a chave de blocagem quanto de comparar os dados. O *Blocking* é encarregado de filtrar cada chave de blocagem criada e gerenciar a necessidade ou não de criar um novo bloco de dados. Em auxílio a esses dados vem o filtro de *Merger* que tem como objetivo evitar a geração de pares redundantes mantendo uma *hashtable* com a combinação de todos os identificadores gerados pelo *ReaderComparator*. Por fim, o *Classifier* é encarregado de verificar os resultados gerados pela comparação feita e classificar os pares de dados como relacionados e não relacionados. Todo esse processamento é feito utilizando mensagens e fluxos rotulados fornecidos pelo *Anthill* para manter a sincronia e escalabilidade entre todas as etapas do processo.

O paralelismo de tarefas foi implementado através de pipelines, os quais eram controlados pelas dependências dos dados. Tendo em vista que o *ReaderComparator* não precisa ler toda a base de dados antes de enviar, ou seja, quando um registro é lido ele já é colocado no pipeline para processamento, foi possível realizar as diferentes tarefas de forma simultânea. Foram exploradas duas abordagens assíncronas. Na primeira, um par de dados é enviado para o *ReaderComparator* logo após a verificação pelo *MergerFilter*. Neste caso, se o mesmo par vem de um bloco diferente, não é enviada novamente. A segunda é vista quando mais de uma instância do mesmo filtro é executada em um mesmo processador de modo assíncrono. Dessa forma, enquanto uma instância utiliza a rede, a outra opera sobre o CPU, e isso apresentou um melhor aproveitamento dos recursos computacionais disponíveis.

3.2.4. Resultados de experimentos

Para realizar os testes foi utilizado um cluster de processadores *AMD Athlon 64 3200 +* com 2GB de memória RAM, conectado por *Gigabit Ethernet*, e rodando Linux 2.6.

Em questão de escalabilidade, os resultados demonstraram que a escalabilidade do algoritmo é quase a mesma para todos os tamanhos de conjuntos de dados, alcançando uma eficiência de acima de 80% para até quinze processadores. Por outro lado, como aumento do número de instâncias, a velocidade tende a diminuir, efeito causado provavelmente pelo custo de comunicação entre os processadores.

Enfim, foi visto que mais de 60% das comparações, ou cerca de 30.000 comparações por segundo, foram realizadas utilizando mensagens que possuíam os *metadados* (mensagem *CompareAlreadyReceivedRecord*). Além disso, cerca de 40% das comparações, ou quase 15 mil comparações por segundo, foram realizadas utilizando registros em bancos de dados locais. Esses números mostram que o algoritmo tem um aquecimento inicial durante o qual existe uma elevada troca de registros, mas depois de algum tempo cai para quase zero.

3.3. Avaliação de técnicas paralelas de blocagem para resolução de entidades e deduplicação

Uma base com dados confiáveis e consistentes é essencial para que seja possível realizar atividades como detecção de fraudes, executar análise e mineração de dados além do *KDD (Knowledge-Discovery in Database)*. Porém o maior problema, e esse vêm se tornando crescente com a popularização da internet, é a falta de padronização dos dados em um banco, como dados inválidos, campos ausentes ou mesmo dados duplicados. Quanto a dados duplicados, é altamente recomendada a identificação dos mesmos para que se possa melhorar a qualidade dos sistemas que o utilizam. O problema de deduplicação consiste em identificar e remover os dados duplicados. Esta tarefa, em uma execução ideal, realiza comparações quadráticas entre todo o volume de dados, o que acaba tornando esse processo altamente oneroso no que se diz respeito a recursos computacionais.

No intuito de diminuir a quantidade de comparações e consequentemente a utilização dos recursos, o trabalho apresentado em [Gonçalvez, et. al. 2008] propõe a utilização de técnicas de blocagem, que agrupam os registros com alguma semelhança com base em métricas básicas. Isso permite que apenas elementos pertencentes ao mesmo grupo sejam comparados. Porém, somente as técnicas de blocagem não foram suficientes para a resolução do problema com uma performance adequada, dessa forma, foi necessário utilizar ambientes paralelos. No trabalho descrito, foram executados testes em cima do ambiente Pareia, a fim de se verificar quais as reais melhoras quando se trabalha em uma arquitetura paralela distribuída.

3.3.1. Ambiente de execução

Foi utilizado o ambiente escalável Pareia juntamente com o arcabouço Anthill. O Pareia fornece uma maneira de separar as etapas do processo de pareamento de um pipeline, fornecendo assim um arcabouço perfeito para que seja possível desenvolver novas metodologias ou mesmo realizar alterações em aspectos que se achem necessário no processo. Já o Anthill explora três possibilidades de paralelismo permitindo assim que tudo seja executado de forma assíncrona, não acarretando em gargalos de desempenho no sistema.

3.3.2. Técnicas de blocagem

Os métodos de blocagem apresentados abaixo, utilizam chaves geradas pelo predicado de blocagem ou uma transformação desse predicado, sempre com a finalidade de agrupar os registros em um bloco comum. O *Standart Blocking*(SB) foi implementado paralelamente no Pareia com a possibilidade de cada filtro estar instanciado em uma maquina diferente e sendo responsável por uma porção das chaves. Isso é garantido através da utilização do *Anthill*. Devido ao *Bigram Blocking*(BI) utilizar apenas a estratégia para geração de chaves diferente do Standart Blocking, a implementação paralela seguiu a mesma estratégia do *Standart Blocking*.A proposta utilizada para que fosse possível paralelizar o processo de blocagem utilizando a *Sorted Neighborhood*(SN) foi particionar os dados de maneira estática em um processo anterior a execução do usuário.

3.3.3. Experimentos e resultados

Os experimentos foram realizados em um cluster de máquinas com processadores *Intel(R) Core (TM)* dois CPU 2.13GHz e 2GB de memória RAM. As métricas utilizadas para a avaliação das blocagens foram as tradicionais Completitude dos Pares (CP) e a Taxa de Redução (TR), além da *F-score* (FS) e a Qualidade dos Pares (QP).

O primeiro ponto a se observar nas três técnicas é a quantidade de pares gerados, e para isso foram executados vários testes com diferentes parâmetros e o resultado apontou que o número de pares gerados pelo BI é aproximadamente o dobro dos gerados pelo SB. Já o SN, a quantidade de pares gerados cresce linearmente de acordo com o tamanho da base. Ainda nesse ponto, foi possível verificar que entre as três técnicas a que possuiu o menor TR é a BI e o maior TR foi obtido pela SN, o que indica que ele foi o mais rápido entre eles.

Em contrapartida, a SN apresentou a menor CP, fato causado pela opção em utilizar parâmetros não tendenciosos. Isso não tira a potencialidade do SN, tendo em vista que ele mesmo acabou por apresentar a melhor taxa de QP. Para uma melhor avaliação, podemos consultar a tabela abaixo e verificar que o SB apresentou o melhor resultado geral.

Técnica de Blocagem	TR (%)	CP (%)	QP (%)
SB	99,41 ± 0,50	60,84 ± 11,00	0,37 ± 0,1
BI	98,72 ± 0,91	69,65 ± 1,93	0,24 ± 0,13
SN	99,94 ± 0,03	19,91 ± 8,04	1,01 ± 0,17

Tabela 1 -Tabela demonstrando os índices alcançados para cada técnica de blocagem

3.4. Comparativo entre trabalhos

Alguns índices puderam ser selecionados para uma comparação entre os trabalhos apresentados e o presente trabalho. A Tabela 2 mostra de forma resumida alguns pontos importantes e em quais trabalhos eles são aplicados ou qual a forma eles são aplicados.

Trabalho	Uso de técnicas de blocagem	Uso de funções de similaridade	Uso de processamento paralelo	Linguagem de programação	Sugestão de algoritmo
P-BSim - 4	Sim	Sim	Sim	JAVA/Scala	Não
[Kawai et. al. 2006] – 3.1	Sim	Não	Sim	JAVA	Sim
[Santos, et. al. 2007] – **Erro! Fonte de referência não encontrada.**	Sim	Não	Sim	C++	Sim
[Gonçalvez, et. al. 2008] – 3.3	Sim	Não	Sim	C++	Sim

Tabela 2 - Comparação entre os trabalhos relacionados e o presente trabalho

Os trabalhos apresentados sugerem abordagens distintas para a resolução do problema de deduplicação de dados.

A diferença entra o presente trabalho (P-BSim) e o *P-Swoosh* [Kawai et. al. 2006] é que o P-BSim visa demonstrar o funcionamento das funções de similaridade em um ambiente paralelo de maneira simples, utilizando ou não dados blocados. O *P-Swoosh* visa uma abordagem genérica para a resolução dos problemas de deduplicação, enquanto o P-BSim tem como objetivo mensurar o ganho real da paralelização de execuções de similaridade e blocagem de dados.

Quanto a [Santos, et. al. 2007], o P-BSim se difere na etapa paralelizada e também por abordar além das técnicas de blocagem a execução de funções de similaridade para a resolução do problema da deduplicação. Em [Santos, et. al. 2007] são propostos alguns modelos para paralelizar a blocagem, porém o P-BSim visou paralelizar apenas a execução das funções de similaridade, partindo de dados que podem ou não estarem blocados.

O trabalho descrito em [Gonçalvez, et. al. 2008] é o que mais se assemelha ao trabalho proposto, com o diferencial que em [Gonçalvez, et. al. 2008] é sugerido um novo algoritmo para o problema de deduplicação e ao invés disso é mensurado o quanto o processo de blocagem juntamente com a paralelização das funções de similaridade beneficia o processo de deduplicação como um todo.

Por fim, o principal objetivo do presente trabalho, P-BSim, não é sugerir um algoritmo para resolução do problema de deduplicação e sim avaliar a execução de funções de similaridade de maneira

paralela sob um conjunto de dados, sendo eles blocados ou não. Para isso se fez necessária a implementação de um processo de blocagem e execução das funções, etapas que são apresentadas com mais detalhes nas próximas seções.

4. P-BSim

Esse capítulo tem como objetivo apresentar o funcionamento do P-BSim (*Parallel Blocking Similarity*), um ambiente que permite a avaliação de desempenho de funções de similaridade executadas de maneira multiprocessadas sob um conjunto de dados blocados. A principal contribuição obtida com o uso desse ambiente foi a de se ter medições significativas quanto à vantagem de realizar o processo de deduplicação de forma paralela, utilizando assim o máximo dos recursos computacionais disponíveis.

O capitulo é dividido da seguinte maneira. Na Seção 4.1 é apresentada uma visão geral da arquitetura implementada para a execução dos testes. Nas Seções 4.2 e 4.3 é apresentado com mais detalhe o funcionamento de cada um dos grupos principais da arquitetura, sendo que na Seção 4.4 é detalhada a forma de implementação das partes focando no quesito técnico. Por fim são apresentados os resultados obtidos após a execução de alguns testes preliminares definidos para esse trabalho.

4.1. Visão geral

A execução do P-BSim pode ser dividida em duas etapas principais conforme apresentado na Figura 1, onde uma delas é a etapa de blocagem de dados e a outra o casamento de dados usando funções de similaridade efetivamente.

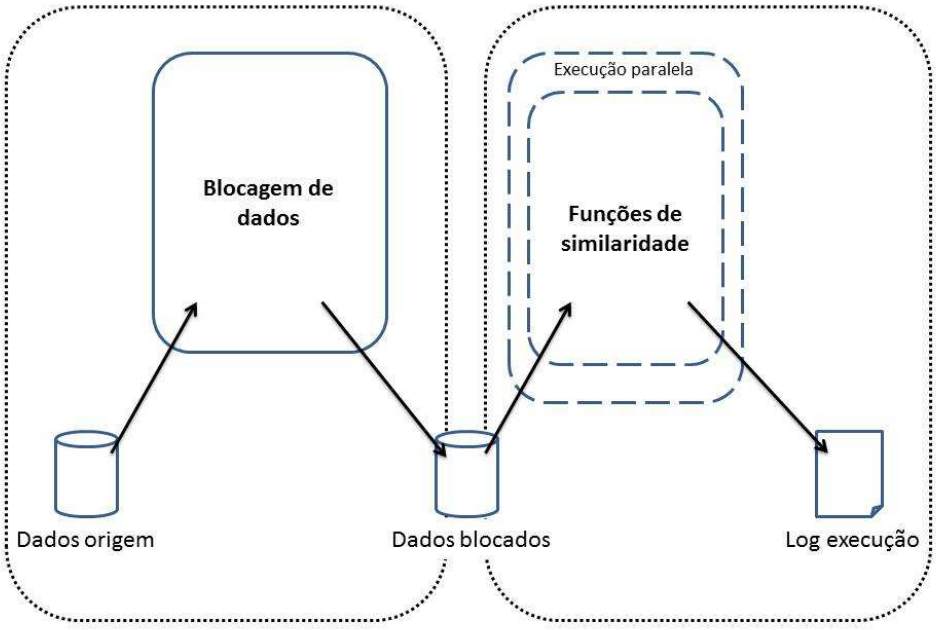


Figura 1 - Visão geral arquitetura P-BSim

Ficam bastante destacadas as duas etapas principais, sendo que na parte de blocagem de dados existe uma interação maior entre bases de dados, enquanto na execução das funções, os dados

blocados são lidos e o resultado da execução das funções para cada registro é armazenada em um arquivo de texto para ser avaliado posteriormente.

4.2. Blocagem de dados

Como visto na revisão bibliográfica, a blocagem de dados é utilizada principalmente para diminuir o número de comparações de dados que não tenham nenhum tipo de relação. Para isso, o processo de blocagem faz a leitura da base de dados origem e com base em uma lógica, gera uma chave, ou um conjunto de chaves, para cada valor e caso um novo registro de dado venha possuir uma chave já registrada, esse registro é inserido nesse bloco.

O processo de blocagem de forma geral é definido conforme a Figura 2, diferenciando das estratégias *Standart Blocking* (ST) e *Bigram Blocking* (BI), utilizadas nesse trabalho, apenas o modo com que é obtida a chave que representa o registro.

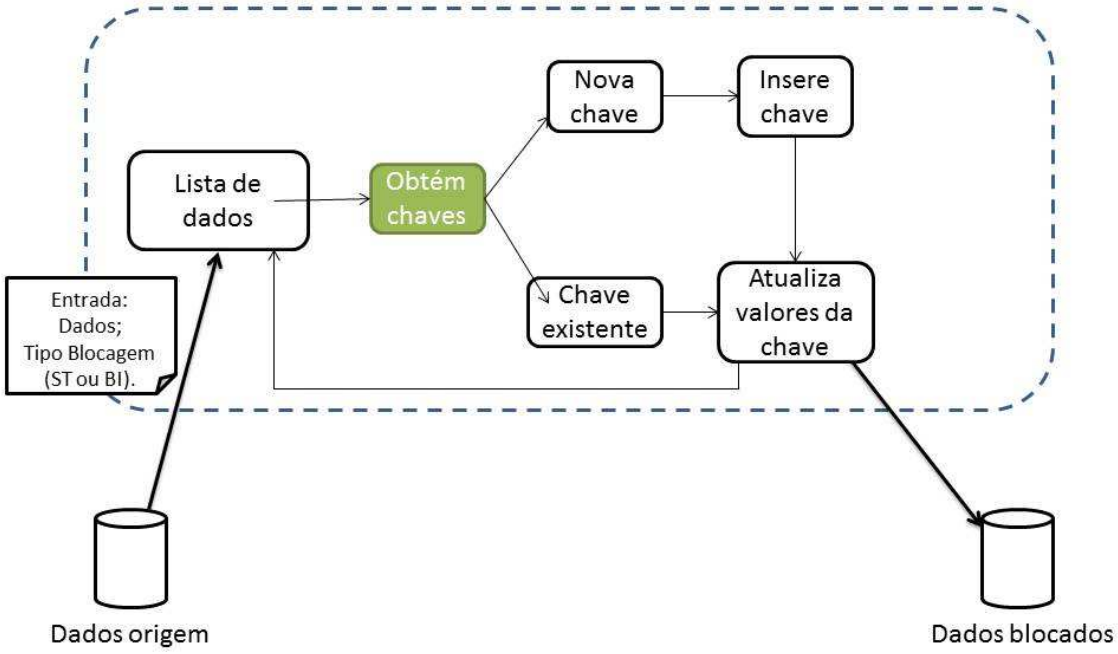


Figura 2 - Esquema representativo de blocagem de dados

De acordo com a Figura 2, que será detalhada a seguir, pode-se ver que a blocagem utilizada foi bastante simples porém funcional. Em resumo, a partir de uma dada base de dados, são lidos os registros que devem ser blocados. Com esses registros em memória, é iniciada uma iteração sobre a lista de dados e para cada valor são realizados os seguintes processos:

- **obtenção de chave:** esse processo foi propositalmente deixado em destaque no esquema, pois ele é a única parte que sofre alteração, dependendo da estratégia de blocagem utilizada no processo. Para esse trabalho o foco foi voltado somente nas estratégias *Bigram Blocking* e *Standart Blocking* (detalhados mais adiante).
- **verificação de chave:** com a chave em mãos é realizada uma verificação para garantir se é uma chave já existente ou não.
- **inserção de uma nova chave:** caso a chave em questão não exista no repositório de dados blocados, ela é inserida no banco de dados tendo como único valor a próprio registro que a gerou.
- **atualização de valores para uma chave:** por fim, caso a chave gerada já exista no repositório, é adicionada uma nova entrada para essa chave com os dados em questão.

Dessa forma, ao final desse processo tem-se uma tabela com duas colunas (Figura 3), onde o índice principal será a chave e os valores serão todas as palavras que compartilham dessa chave. Os valores apresentados na Figura 3 fazem referencia a um processo de blocagem utilizando a estratégia *Bigram Index*.

	chave [PK] character varying(255)	valores text
756	ABBEEIGOIAIROIR:RA	R:GOIABEIRA#
757	ABBEEIGUIRIROBRAROUA	GUABIROBEIRA#
758	ABBEEL	ABEL SCUISSIATO#ABEL SCUISSIATO#ABEL AMARAL DOS SANTOS#
759	ABBEELETIZLITEZA	RUA ELIZABETE GULART#ELIZABETE GULART#
760	ABBEELETIZLIZA	ELIZABET GOULART#
761	ABBEELISSA	PRINCESA ISABEL#PRINCESA ISABEL#PRINCESA ISABEL#PRINCESA ISABEL#PRINCESA ISABEL#I
762	ABBEELIZZA	PRINCESA IZABEL#PRINCESA IZABEL#PRINCESA IZABEL#LINHA SANTA IZABEL#IZABEL OSOSKI-
763	ABBEELLLLORA	JOAO RABELLO COUTINHO#
764	ABBEELLORA	JOÃO RABELO COUTINHO#
765	ABBEEMIVRIVA	MARIA BERTOJA RIVABEM#LUIZ RIVABEM#
766	ABBEENSTTA	VEREADOR AUGUSTO STABEN#
767	ABBEERRT	ABERT EINSTEIN#
768	ABBICOILISLINTONSTTATA	DOS CONTABILISTAS#
769	ABBIEIGUIRIROBRAROUA	GUABIROBEIRA#
770	ABBIFAIO	RUA FABIO DE SOUZA#FABIO FANUCHI#
771	ABBIGUIROTROTUUAUB	RUA GUABIROTUBA#GUABIROTUBA#
772	ABBIHOILLH	ABILHO OHLZNANN#ABILHO OHLZNANN#ABILHO MONTEIRO#
773	ABBIIÁSA	SABIÁ#LINHA RIO SABIÁ#
774	ABBIIÁSAÁS	DAS SABIÁS#
775	ABBIILIOLI	ABILIO RAMOS#ABILIO RAMOS#ABILIO HOLZMANN#ABILIO DA COSTA SOARES#ABILIO DA COSTA
776	ABBIILLOON	ABILON DE SOUZA NEVES#ABILON DE SOUZA NAVES#ABILON DE SOUZA NAVES#
777	ABBÍIOLIÍL	RUA ABÍLIO HOLZMANN#ABÍLIO PEIXOTO#ABÍLIO HOLZMANN#
778	ABBIIRITRATA	RUA ITABIRA#ITABIRA#ITABIRA#ITABIRA#ITABIRA#ITABIRA#ITABIRA#ITABIRA#ITAB

Figura 3 - Exemplo do resultado de uma blocagem utilizando *Bigram Blocking*

4.2.1. Standart Blocking

A blocagem padrão, referenciada na literatura como *Standard Blocking* [Christen 2007] (SB) , é uma técnica tradicional e simples, que agrupa os registros de acordo com uma chave de blocagem predefinida. Nesse trabalho ela foi implementada de acordo com o esquema representado no Figura 4.

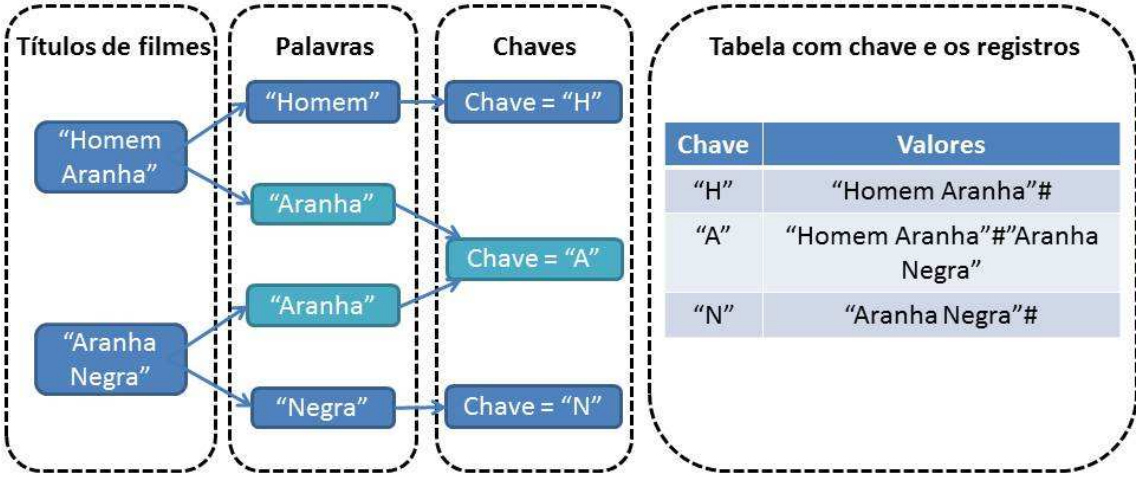


Figura 4 - Esquema *Standart Blocking*

No P-BSim, o processo de blocagem utilizando a técnica *Standart Blocking* define como o predicado de blocagem (chave) o primeiro caractere de cada palavra do registro. Por exemplo, para uma lista de títulos de filme, como mostrada na Figura 4, é analisado cada filme e obtida a chave para cada uma das palavras que compõe o título como um todo. Por exemplo, o filme "Homem Aranha" possui duas chaves que o representam para futuras comparações, sendo elas os caracteres "H" e "A". Por sua vez, para o filme "Aranha Negra", são geradas mais duas chaves, os caracteres "A" e "N", porém no momento da verificação de existência da chave "A" no repositório de dados blocados, é visto que "A" já é uma chave existente. Dessa forma, os valores da chave "A" são atualizados e essa passa a contar com os títulos "Homem Aranha" e "Aranha Negra".

Fica claro que o número de chaves para um determinado registro vai ser igual a sua quantidade total de palavras. Foi observado que, para registros com um maior número de palavras que, consequentemente geravam mais chaves, a chance dos dados estarem em um grupo relacionado era bem maior mesmo se existissem erros de grafia em alguma das palavras. Dessa forma, podem-se agrupar de maneira simples os dados que tenham alguma relação mínima entre si. Casos como erro de grafia passam despercebidos nessa estratégia, Porém, a análise do repositório de dados blocados após a conclusão de todo processo mostrou que, mesmo tendo falhas, se torna muito mais viável realizar a execução das funções de similaridade a partir de dados blocados dessa forma do que trabalhar com o volume de dados inteiro executando as funções como um produto cartesiano (PC).

4.2.2. Bigram Blocking

O método *Bigram Blocking* (BI) [Baxter et al. 2003, Bilenko et al. 2006] permite uma abordagem difusa da blocagem (*fuzzy blocking*), possibilitando que alguns erros tipográficos possam ser captados. No P-BSim, a estratégia de blocagem *Bigram Blocking* foi implementada para suportar não somente a geração de chaves pelos bigramas, mas como por qualquer n-gramas que são enviados como parâmetro no momento da execução. Na Figura 5, é mostrado o esquema de funcionamento do algoritmo, partindo do momento em que é feita a leitura do dado, sua blocagem e geração de chaves através de seus n-gramas.

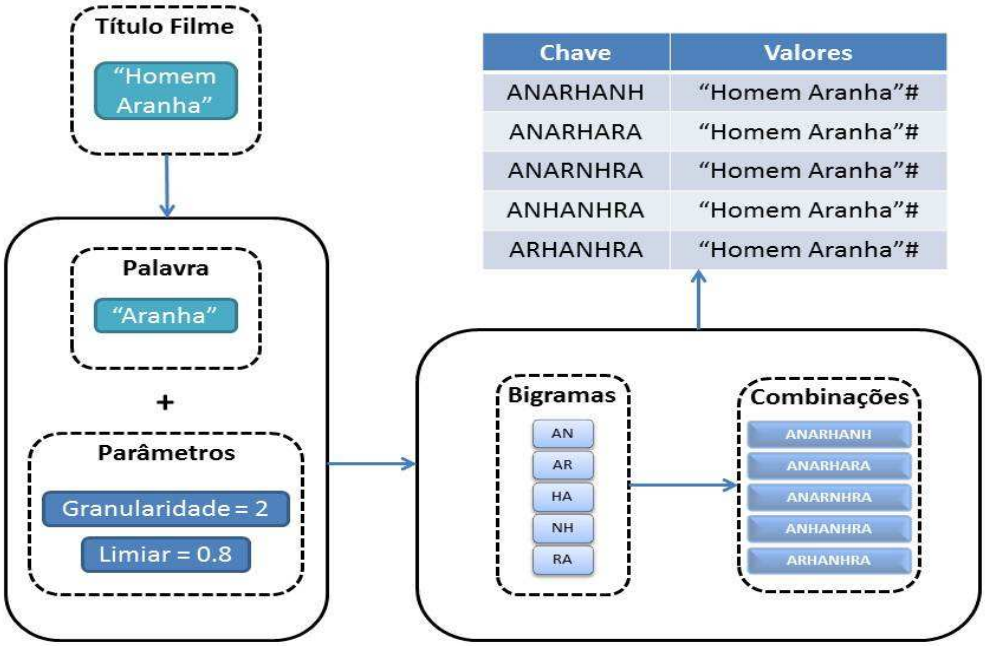


Figura 5 - Funcionamento do algoritmo utilizando a estratégia *Bigram Blocking*

No exemplo acima, tem-se novamente o título do filme "Homem Aranha" e deseja-se blocar esse valor de acordo com a estratégia em questão. Igualmente à estratégia *Standart Blocking,* o processo é feito para cada palavra do dado em questão, para aumentar as chances de se terem dados que representem a mesma entidade no mesmo grupo. Com a palavra já separada, são consultados dois parâmetros importantes para a execução do algoritimo, sendo eles a granularidade, que indica qual será o valor de n que o algoritmo deve gerar. Por exemplo, 2 para bigramas, 3 para trigramas e assim por diante; e o limiar é uma necessidade da própria estratégia de blocagem, o limiar ou *threshold*, um valor que varia entre zero e um e define o tamanho da combinação entre todos os n-gramas gerados na execução do algoritmo. Por exemplo para um limiar igual a 0.9, dado um registro que tenha 10 bigramas, serão feitas todas as combinações possíveis entre 9 bigramas para esse conjunto. Esse é um

valor importante, pois quanto menor o valor informado, menor será a combinação gerada e consequentemente maior o número de chaves para o registro em questão.

Seguindo a execução, com base na palavra, juntamente com os parâmetros são obtidos todos os n-gramas (nesse caso bigramas) possíveis para a palavra, ordenados alfabeticamente. Após isso, com base na quantidade de bigramas gerados multiplicados pelo valor do limiar, é obtido o tamanho que as combinações devem possuir. Por exemplo para a palavra “casa”, temos os bigramas {“ca”, “as”, “sa”}. Com um limiar igual a 0.9, é aplicada uma expressão simples (tamanho da palavra multiplicado pelo limiar), e o resultado indica o tamanho da combinação, para esse caso 3,6 arredondado para 3. Então são geradas todas as combinações entre 3 elementos para os 3 bigramas gerados resultando na seguinte combinação {“ascasa”}. Por fim, cada uma das combinações entre os bigramas representa uma chave para aquele valor, e é feita a inserção no repositório de dados juntamente com o valor completo, no caso de nosso exemplo, “Homem Aranha”.

Esse é um processo que exige muito recurso computacional, principalmente quando se trabalha com dados que possuem um número grande de palavras. Porém é a estratégia que melhor agrupa os dados em categorias semelhantes.

4.3. Processamento paralelo de funções de similaridade

As funções de similaridade têm se tornado cada vez mais importantes no processo de deduplicação de dados. O problema é que muitas vezes o processo de comparação realizado por essas funções demandam excessivo recurso computacional, fazendo com que, em muitos casos, uma abordagem comum, onde o processamento é realizado em sequencia, seja inviável em um volume de dado extenso.

O P-BSim implementa a abordagem de paralelização no processamento das funções de similaridade, fazendo com que seja possível distribuir a execução da função em diferentes processos, alimentando o mesmo arquivo de *log* com os resultados das comparações. Isso torna viável a exploração dos múltiplos núcleos que os novos processadores vêm oferecendo. O funcionamento básico de toda a estrutura envolvida na execução das funções é representado na Figura 6 e detalhado nos próximos parágrafos.

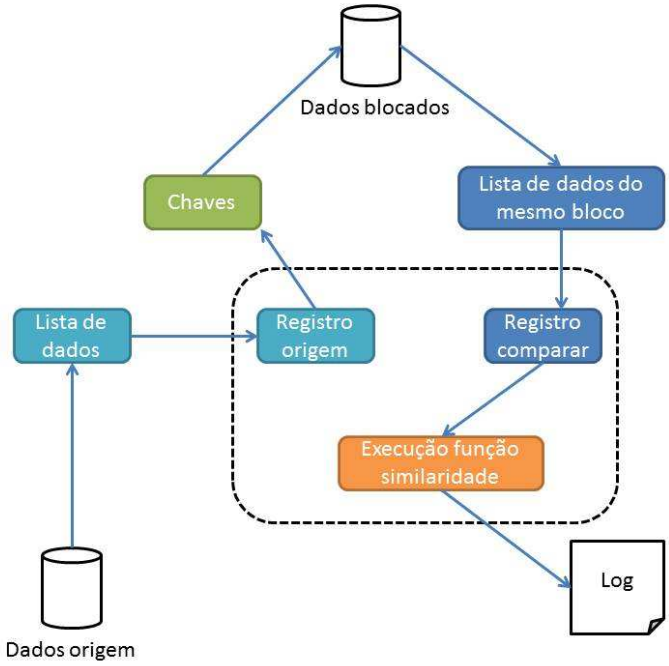


Figura 6 - Execução das funções de similaridade

No P-BSim, as funções de similaridade podem ser executadas utilizando-se ou não os dados que foram blocados. Para os casos de uso dos dados blocados, o processo de blocagem em cima da base de dados avaliada já deve ter sido realizado de forma independente. Essa abordagem foi necessária para que fosse possível avaliar o quanto a blocagem dos dados influenciaria no resultado final tanto em questão de desempenho quanto de qualidade.

O algoritmo começa recebendo uma lista contendo os dados que devem ser comparados a fim de verificar problemas de duplicidade. Partindo dela, para cada registro dessa lista tem-se a opção de gerar as chaves para consulta de dados blocados tanto com a estratégia *Strandart Blocking* quanto com a *Bigram Blocking*, ou simplesmente realizar a comparação desse registro com todos os outros.

Quando se deseja utilizar os dados blocados, cada registro deve gerar uma ou mais chaves e com isso o algoritmo consulta o repositório de dados blocados, variando conforme a estratégia de blocagem escolhida. A consulta nesse repositório retorna uma lista com os registros que possivelmente representam a mesma entidade no mundo real. Essa lista é percorrida e as funções de similaridade são executadas.

A questão de paralelização desse processo é explicada com maior detalhes na seção de implementação, pois grande parte da complexidade desse modelo de programação é abstraída pela API utilizada. Isso fez com que o grau de complexidade na implementação do P-BSim caísse bastante, permitindo que durante todo o processo todos os núcleos do processador estivessem sendo utilizados.

Por fim, no período de execução das funções, para cada par de registros é gerado um escore que representa quão similares eles são. Como um dos objetivos principais do processo de deduplicação é localizar dados semelhantes, no P-Bsim eles são armazenados no arquivo de *log* final com informações sobre comparações que tiveram um escore maior do que um limiar informado no momento da

execução do processo. Esse arquivo de *log* é salvo no próprio computador no qual o P-BSim está rodando.

4.4. Implementação

Esta seção apresenta as tecnologias utilizadas para o desenvolvimento do P-BSim, bem como a descrição do processo de implementação do projeto. O P-BSim teve seu desenvolvimento dividido em duas etapas, sendo elas a implementação de toda a lógica de blocagem e posteriormente a lógica para execução das funções de similaridade. Na parte de blocagem, a implementação foi toda feita utilizando a linguagem JAVA juntamente com banco de dados Postgres SQL. Já na parte de execução das funções de similaridade, foi utilizado a linguagem de programação SCALA, armazenando os resultados em um arquivo de *LOG* gerenciado pela mesma.

4.4.1. Blocagem no P-BSim

A Figura 6 representa o digrama de classes do módulo de blocagem. Através dele podemos entender melhor o funcionamento da etapa de blocagem utilizada no P-BSim.

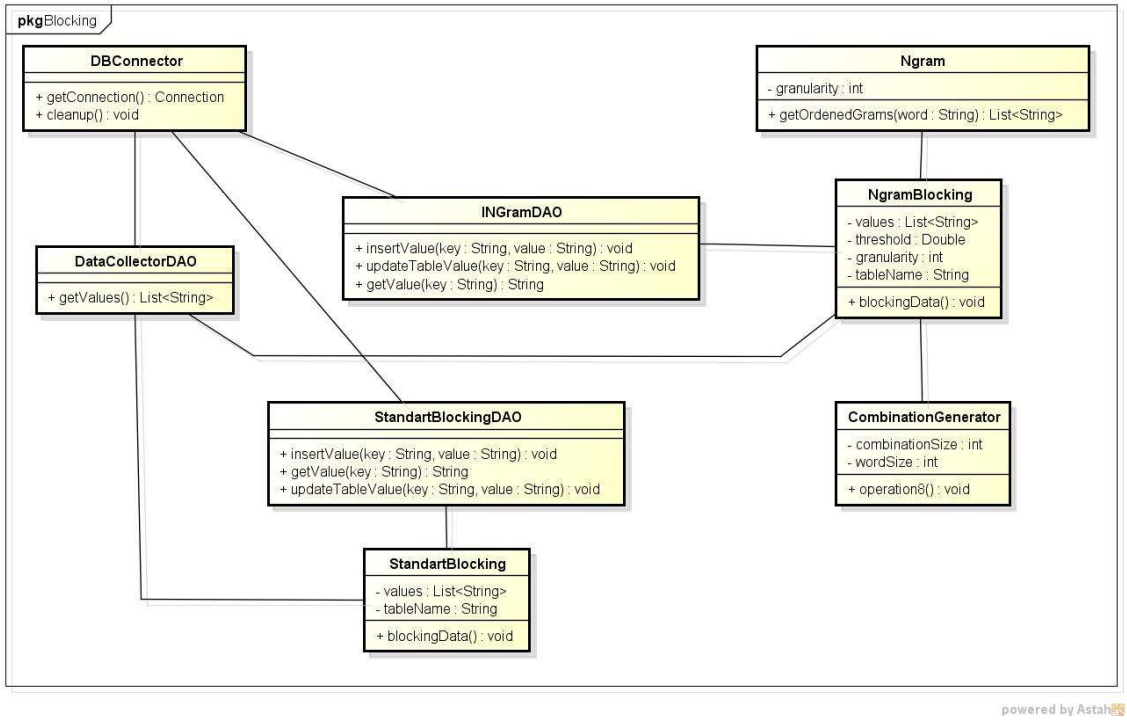


Figura 7 - Digrama de classes módulo de blocagem P-BSim

A classe *DBConector* é a responsável por fornecer acesso às tabelas do banco de dados. Para garantir um melhor aproveitamento das conexões com o banco, foi utilizado o framework *DBPool*[5], o qual possibilita trabalhar com um *pool* de conexões com o banco em aplicações *stand-alone*. Cada uma

[5] Disponível em: http://www.snaq.net/java/DBPool/

das estratégias de blocagem conta com um DAO *(Data Acces Object),* classes que são encarregadas de realizar as transações com o banco. As transações podem ser resumidas em: inserir uma nova chave com um possível valor, atualizar os valores de uma chave e por fim obter todos os valores de uma determinada chave.

Os métodos de inserção de chave com valor, atualização de valores da chave e obtenção de valores, são utilizados por ambas as estratégias de blocagem e mantém o resultado do processo em uma tabela (Figura 8) no banco de dados. Essa tabela possui apenas duas colunas, sendo uma delas para o armazenamento da chave e a outra para armazenamento de todos os possíveis valores para a chave em questão.

chave	valores
'SCEORR'RCRE	Harry Potter and the Sorcerer's Stone#Harry Potter and the Sorcerer's Stone#Harry Potter and the Sorcerer's Stone#
'SCEORR'RCSO	Harry Potter and the Sorcerer's Stone#Harry Potter and the Sorcerer's Stone#Harry Potter and the Sorcerer's Stone#
'SCEORR'RESO	Harry Potter and the Sorcerer's Stone#Harry Potter and the Sorcerer's Stone#Harry Potter and the Sorcerer's Stone#
'SCEORRCRESO	Harry Potter and the Sorcerer's Stone#Harry Potter and the Sorcerer's Stone#Harry Potter and the Sorcerer's Stone#
'SCER'RCRESO	Harry Potter and the Sorcerer's Stone#Harry Potter and the Sorcerer's Stone#Harry Potter and the Sorcerer's Stone#
'SCHE'HAIELI	Charlie's Angels: Full Throttle#Charlie's Angels#Charlie's Angels: Full Throttle#Charlie's Angels: Full Throttle#Charlie's Angels#Charlie's Angels 2 Full
'SCHE'HAIERL	Charlie's Angels: Full Throttle#Charlie's Angels#Charlie's Angels: Full Throttle#Charlie's Angels: Full Throttle#Charlie's Angels#Charlie's Angels 2 Full
'SCHE'HALIRL	Charlie's Angels: Full Throttle#Charlie's Angels#Charlie's Angels: Full Throttle#Charlie's Angels: Full Throttle#Charlie's Angels#Charlie's Angels 2 Full
'SCHE'IELIRL	Charlie's Angels: Full Throttle#Charlie's Angels#Charlie's Angels: Full Throttle#Charlie's Angels: Full Throttle#Charlie's Angels#Charlie's Angels 2 Full
'SCHHAIELIRL	Charlie's Angels: Full Throttle#Charlie's Angels#Charlie's Angels: Full Throttle#Charlie's Angels: Full Throttle#Charlie's Angels#Charlie's Angels 2 Full
'SCOCTECLELLOLOR	Anger Management Collector's Edition#Final Fantasy Collector's Edition#Harry Potter And The Philosophers Stone Widescreen Collector's Edition#
'SCOCTECLELLOLR'	Anger Management Collector's Edition#Final Fantasy Collector's Edition#Harry Potter And The Philosophers Stone Widescreen Collector's Edition#
'SCOCTECLELLOLTO	Anger Management Collector's Edition#Final Fantasy Collector's Edition#Harry Potter And The Philosophers Stone Widescreen Collector's Edition#
'SCOCTECLELLORR'	Anger Management Collector's Edition#Final Fantasy Collector's Edition#Harry Potter And The Philosophers Stone Widescreen Collector's Edition#

Figura 8 - Exemplo de tabela com os dados blocados

Quanto à lógica para a execução da blocagem, no momento da execução é selecionado qual a estratégia deve ser usada para a geração das chaves. Caso seja adotada a estratégia *Standart Blocking* é utilizada uma instância da classe *StandartBlocking*, a qual tem seu funcionamento representado no diagrama de atividades mostrado na Figura 9 que foi explicado em detalhes na seção 4.2.1.

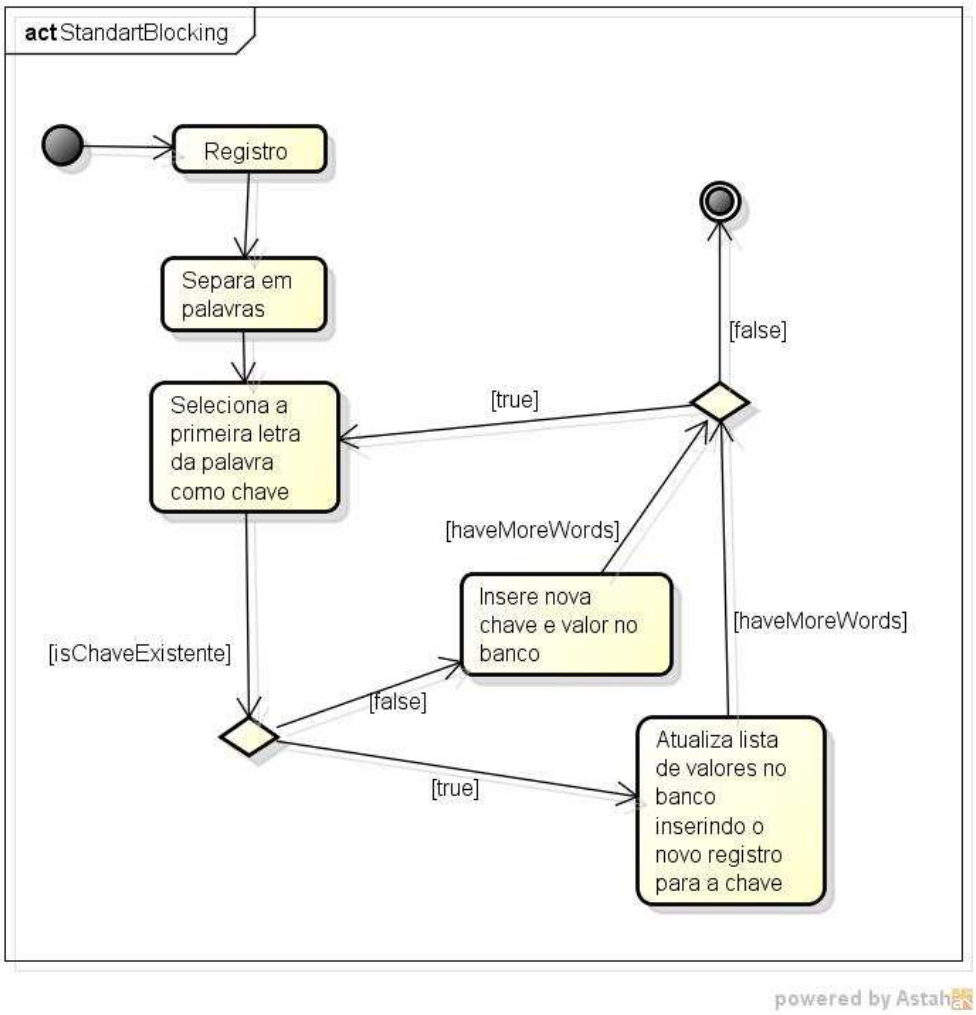


Figura 9 - Diagrama de atividades *Standart Blocking*

Para a estratégia *Bigram Blocking* é utilizada uma instância da classe *NGramBlocking,* a qual recebe como parâmetros no momento da execução a quantidade de *grams* que o algoritmo irá trabalhar, assim como o *threshold* que define o tamanho das combinações de *ngrams* que será usada como chave. A lógica de funcionamento do *NGramBlocking* pode ser vista no diagrama de atividades que está na Figura 10, explicada na seção 4.2.2.

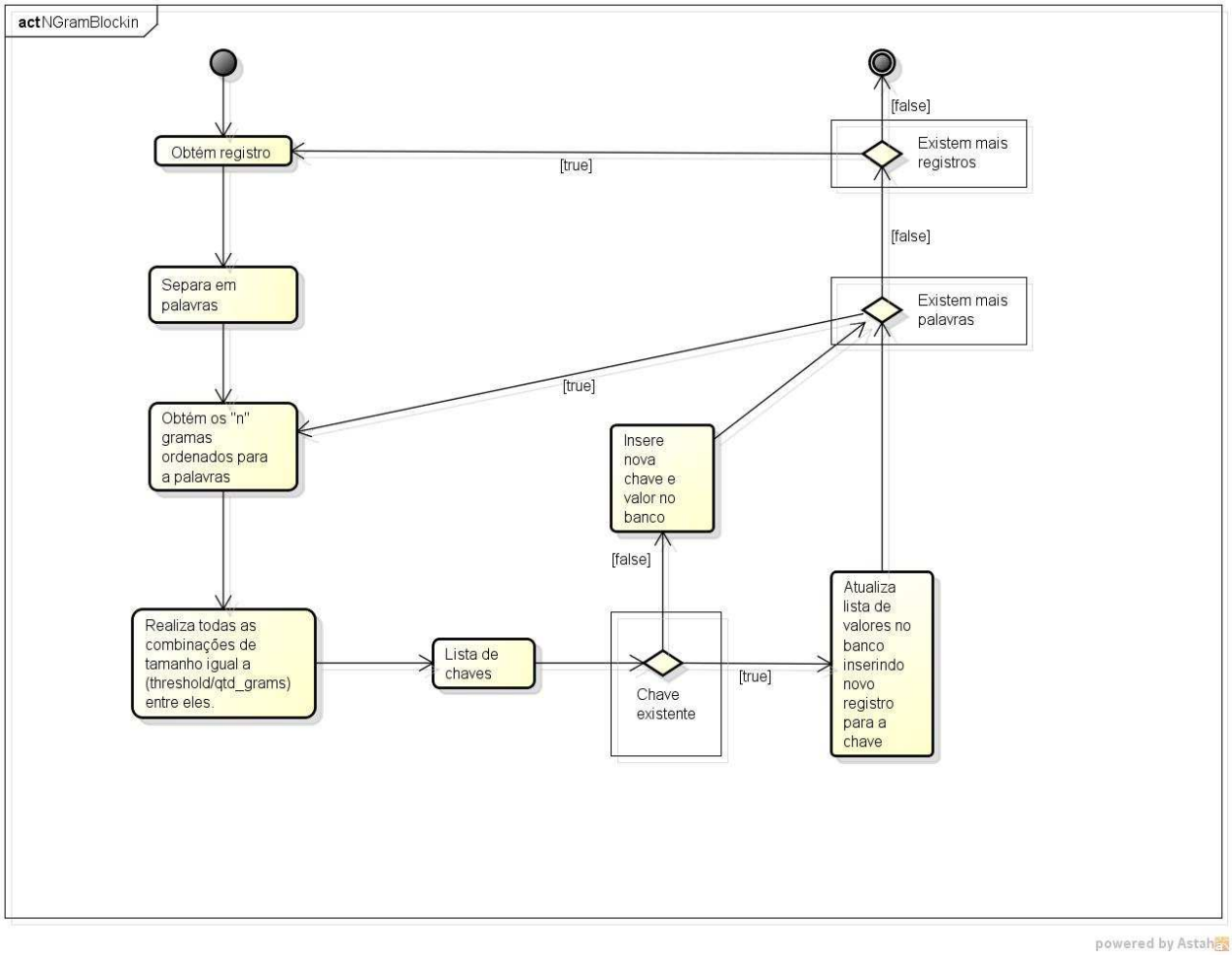


Figura 10 - Diagrama de atividades do *NGramBlocking*

4.4.2. Funções de similaridade no P-BSim

A segunda etapa, onde foi desenvolvida toda a lógica para a execução das funções de similaridade de forma paralela, foi desenvolvida em Scala, utilizando todo o conjunto de artefatos JAVA produzidos na etapa anterior. Em termos de arquitetura de software essa é uma etapa mais simples graças a uma abstração fornecida pela linguagem Scala.

A Figura 11 apresenta o digrama de classes desse módulo, partindo dos artefatos já existentes na primeira etapa e concluindo na geração de um arquivo de *log.*

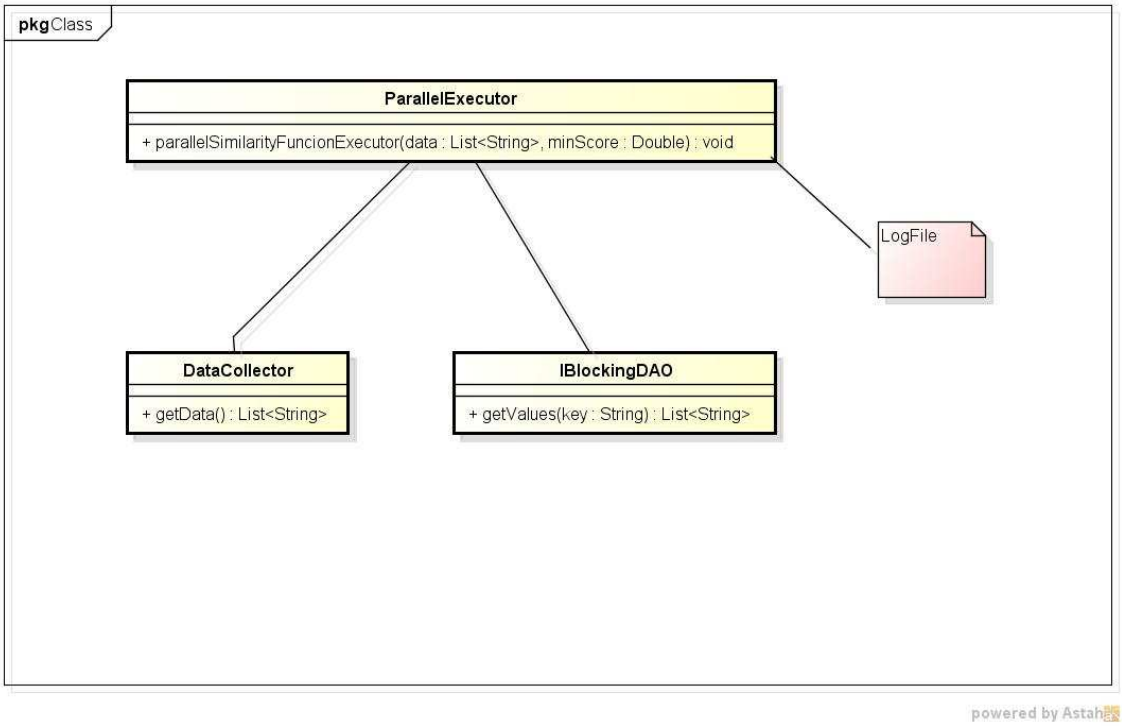


Figura 11 - Diagrama de classes - Execução de funções de similaridade

Esse módulo é composto apenas por uma classe principal que faz consulta utilizando as classes em JAVA já implementadas na etapa de blocagem. Isso é possível, pois o SCALA fornece acesso a código JAVA de forma transparente (Mais informações sobre a linguagem de programação SCALA podem ser vistas nas referências [Odersky et. al. 2008]. Esse acesso transparente à linguagem JAVA permitiu a utilização da biblioteca *SIMMETRICS*[6], uma biblioteca em JAVA que implementa diversas funções de similaridades. Dessa forma, foi possível realizar os testes com diferentes funções.

A classe *ParallelExecutor* recebe uma lista de dados e um valor que indica qual o menor valor de escore para que uma comparação entre no arquivo de *log.* Na execução do processo de comparação utilizando as funções de similaridade, a lista de valores é convertida para uma lista paralela, ou *parallel collections*: uma implementação que permite iterar sobre essa lista de maneira paralela e consequentemente toda a ação sobre essa iteração é realizada de maneira paralela que por padrão utiliza todos os núcleos de processadores disponíveis na arquitetura que está sendo utilizada. Dessa forma, foi possível paralelizar o processamento das funções de similaridade de forma transparente, deixando o controle de transações para o *SCALA Run Time*.

Como nos testes realizados utilizando o P-BSim foram utilizados dados blocados e não blocados, a implementação dos métodos que realizavam as comparações entre os dados sofreu alguma alteração para esses dois cenários, tendo em vista que, para testes onde não era necessário utilizar dados blocados, a comparação foi feita como um produto cartesiano, não sendo necessária a preocupação com geração de chaves e resgate de valores pertencentes ao mesmo grupo.

[6] Disponível em: http://sourceforge.net/projects/simmetrics/

A Figura 12 e a Figura 13 exemplificam o funcionamento dos algoritmos de comparação utilizando as funções de similaridade, com dados blocados (Figura 12) e sem dados blocados (Figura 13). Visando garantir uma performance perfeita para a execução paralela, antes de iniciar a comparações dos objetos quando blocados, é carregado um cache que armazena todo o repositório de dados blocados em um *ParMap* [7] para o caso da execução paralela ou um *Map* [8] para a execução sequencial.

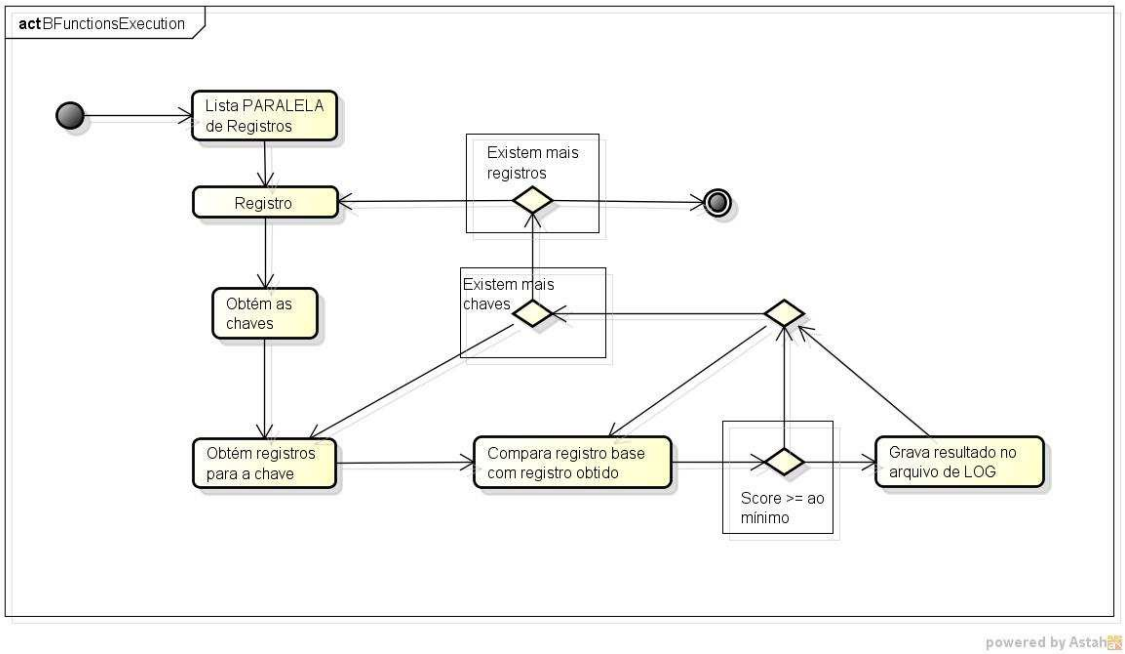


Figura 12 - Execução de funções de similaridade com dados blocados

A Figura 12 exemplifica o funcionamento das funções de similaridade quando os dados em questão já estão blocados. Levando em conta que os dados a serem avaliados já foram blocados, é feita uma iteração nessa *parallel collection* a fim de realizar a comparação de cada palavra.

Para cada registro, são extraídas todas as chaves dependendo do modo de blocagem escolhido. Para cada chave, são obtidos todos os valores que façam parte desse grupo, e por fim a comparação entre o registro da lista principal com o registro do bloco é realizada. O resultado dessa função é verificado, e se for maior ou igual a um e*score* que é enviado como parâmetro, é gravado no *LOG* os registros e o escore. Esse processo é feito até a lista principal ser totalmente percorrida. A escrita no arquivo de log foi implementada utilizando a biblioteca *loj4j*[9] disponibilizada pela Apache Foundation. Porém, foi necessário utilizar um *appender*, objeto que realiza a escrita assíncrona no arquivo. Dessa

[7] Estrutura de dados paralela disponibilizada pelo Scala: http://www.scala-lang.org/archives/downloads/distrib/files/nightly/docs/library/index.html#scala.collection.parallel.immutable.ParMap$

[8] Objeto disponibilizado pelo Scala que provê uma estrutura de dados com a operação de um *Map sequencial: http://www.scala-lang.org/archives/downloads/distrib/files/nightly/docs/library/index.html#scala.collection.Map$*

[9] Disponível em: http://logging.apache.org/log4j/2.x/

maneira, a execução do algoritmo, principalmente quando realizado em paralelo, não sofreu com perda de performance normalmente causada quando ocorre uma operação de I/O (leitura ou escrita de dados em disco).

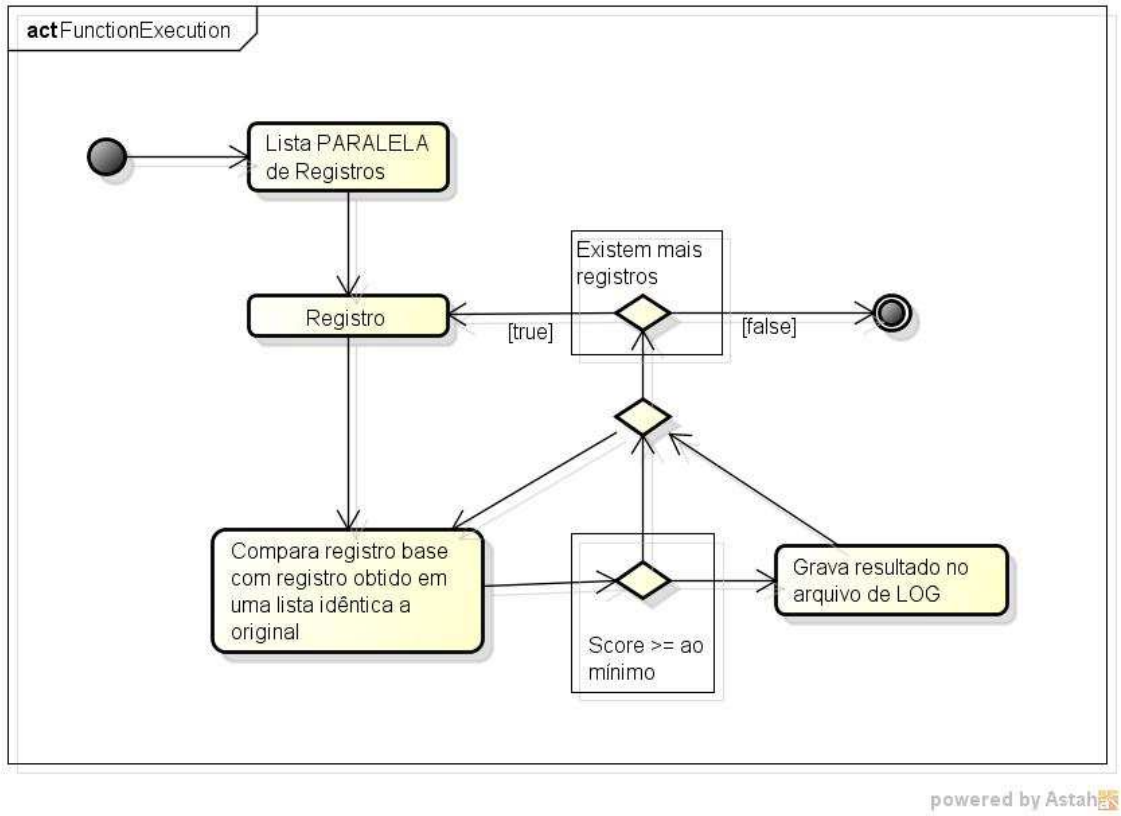


Figura 13 - Execução de funções de similaridade com dados simples

Para a execução das funções em dados que não foram blocados, é feito um produto cartesiano (PC). A "*parallel collection"* onde estão os dados a serem avaliados é clonada e cada registro é comparado com todos os registros da lista. O filtro é idêntico ao utilizado para dados blocados, leva em conta um escore mínimo recebido como parâmetro e verifica para cada execução se o escore resultado é maior ou igual. Caso o escore retornado pela função for um valor maior ou igual ao limiar, as informações são gravadas no LOG.

A execução das funções em ambiente não paralelo opera sob essa mesma lógica, porém não são utilizada as estruturas de dados disponibilizadas pela *parallel collection.* Dessa forma, a iteração sobre a lista é realizada de maneira sequencial. Essa implementação foi feita para que fosse possível ter resultados reais de performance afim de efetuar as comparações no final do trabalho.

No arquivo do LOG, além do resultado das funções que passaram pelo filtro de escore é gravado também a data e hora de início e fim do processo. Dessa forma, é possível realizar análises de desempenho temporal entre as diferentes abordagens

4.5. Experimentos

O P-BSim foi concebido com o objetivo principal de viabilizar a execução de experimentos para se chegar a resultados demonstrando qual o real impacto que a abordagem de programação paralela juntamente com o processo de blocagem de dados tem no processo de deduplicação.

Toda a bateria de testes foi executada em um computador possuindo o seguinte *hardware*: processador *Intel Core i5 2410M* (quatro núcleos) com *4GB* de memória *RAM*. Quanto aos dados, o tamanho das amostragens que seriam utilizadas para a execução da blocagem e das funções de similaridade partiu do seguinte princípio: partindo de uma massa de dados X, contendo Y registros, os dados eram divididos em cinco grupos. Cada grupo utilizava uma amostra de dados da tabela original contendo a seguinte a proporção: de 1/10 no grupo 1, 1/4 no grupo 2, 1/2 no grupo 3, 3/4 no grupo 4 e 1 no grupo 5. Dessa forma foi possível montar repositórios com um volume de dados variando desde um valor mínimo, até todos os registros do repositório de dados original. Quanto ao parâmetro de entrada, como apresentado na seção anterior, o único parâmetro que o algoritmo leva em consideração é o limiar que a função deve retornar para que a comparação seja catalogada no log. O valor foi fixado em 0.9 para todos os testes. Por fim, quanto às funções de similaridade utilizadas, foram selecionadas as seguintes funções dentro das disponíveis na biblioteca *Simetrics: Levenshtein, BlockDistance, EuclideanDistance* e *Jaro.* Todas as funções foram executadas ao mesmo tempo, dentro do mesmo *"loop"* de execução, para que fosse possível tirar maior proveito do paralelismo.

Foram utilizados diferentes domínios de dados, que vão desde nomes de cidades até nomes de instituições de ensino. Cada uma das bases avaliadas sofreu uma análise antes da execução dos testes para que, com o conhecimento do domínio, fosse possível chegar em conclusões sobre quanto o tipo de dados avaliado interfere na execução das funções de similaridade. Isso viabilizou a obtenção de resultados em cenários bastante heterogêneos, tendo em vista que, para domínios diferentes, a execução do algoritmo pode ser mais trabalhosa ou não.

RESULTADOS

Tendo os artefatos devidamente desenvolvidos, deu-se início aos testes preliminares, que foram separados em quatro grandes grupos. Sendo eles:

- dados não blocados sem a utilização de programação paralela
- dados não blocados utilizando programação paralela
- dados blocados utilizando programação paralela
- dados blocados sem a utilização de programação paralela.

Os testes também foram separados em grupos tendo como critério para separação o tipo de dado avaliado. Dessa forma, foi possível avaliar os resultados de forma individual e, realizar um comparativo entre os diferentes tipos de testes a fim de verificar que tipo de impacto diferentes tipos de dados provocaram. Os testes foram classificados da seguinte forma:

- **Teste 1 (Teste cid.)** – testes realizados sob uma base contendo nome de cidades brasileiras. Antes da execução foi realizada uma pequena auditoria na base através de consultas *SQL* e visto que existiam diversos dados repetidos e erros léxicos. No total, a base possuía 10830 registros, os quais foram divididos conforme o fator de amostragem de dados explicado a cima.
- **Teste 2 (Teste rua)** – testes realizados sob uma base de dados contendo nome de ruas. Na verificação realizada nos dados foi visto que os nomes de ruas normalmente são "registros" maiores, contendo 3 ou mais palavras. Uma inconsistência recorrente nessa base era a denominação da via, onde em alguns registros estava apenas "R", outro "R.", outros "Rua" e outras descrições para identificar que a via era uma rua comum. O mesmo ocorria para avenidas, travessas, servidões e outras denominações de vias. No total a base possuía 9895 registros.
- **Teste 3 (Teste inst.)** – teste realizado tendo como entrada de dados nomes de instituições de ensino brasileiras, sendo elas Universidades, Escolas técnicas, colégios e alguns outros tipos de instituições. Na verificação de domínio do dado foi visto que os registros dessa tabela eram os maiores se comparados com os utilizados nos outros testes (1 e 2) porque normalmente antes do nome, era indicado o tipo da instituição de ensino. Uma das inconsistências que foram notados foi justamente a denominação do tipo de instituição, que era escrita de diversas formas como por exemplo "ESC", "ESC.", "ESCOLA", entre outras para identificar que era uma escola. Essa base possuía um número menor de registros, totalizando em 7832 entradas de dado na base.

Para uma melhor organização do trabalho, os resultados são apresentados separados em seções e, ao final, é feito um comparativo entre todos os resultados obtidos.

4.5.1. Dados não blocados sem a utilização de programação paralela

Este pode ser considerado o cenário mais comum, onde os dados não foram blocados e o processo de comparação também não foi implementado da maneira paralela. Nessa bateria de teste foi notado que a carga do processador variava entre 30% e 50%, normalmente utilizando apenas um dos quatro núcleos disponíveis no processador.

Para dados que não foram blocados, a execução das funções de similaridade ocorre como um produto cartesiano, onde um registro é comparado com todos os outros da lista de dados. Esse caso, teoricamente, é o pior dos casos em um processo de deduplicação, o que exige mais recursos e costuma ser o mais demorado. A partir daqui, sempre que for feita uma referencia a produto cartesiano está se arremetendo a comparação de dados não blocados.

Ao final dessa etapa, para os três testes definidos, foram obtidos os resultados apresentados na Figura 15 para nome de cidades, na Figura 16 para nome de ruas e na Figura 17 para nome de instituições de ensino.

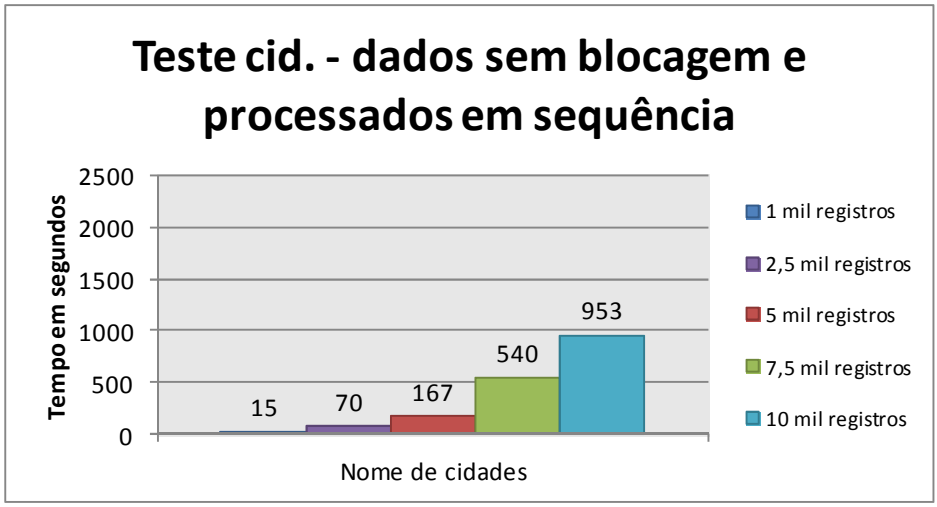


Figura 15 – Teste em nomes de cidades – tempo de execução para dados sem blocagem e sem paralelização da execução

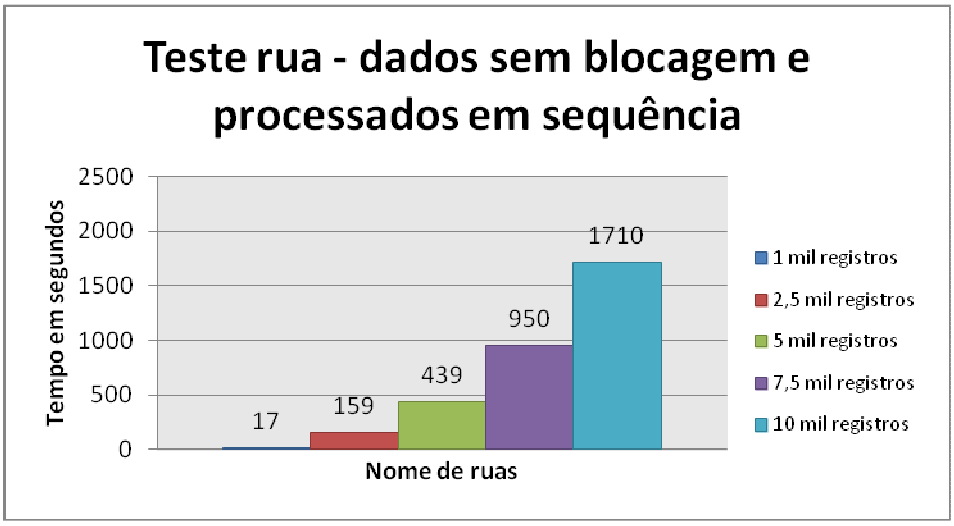


Figura 16 - Teste em nomes de ruas – tempo de execução para dados sem blocagem e sem paralelização da execução

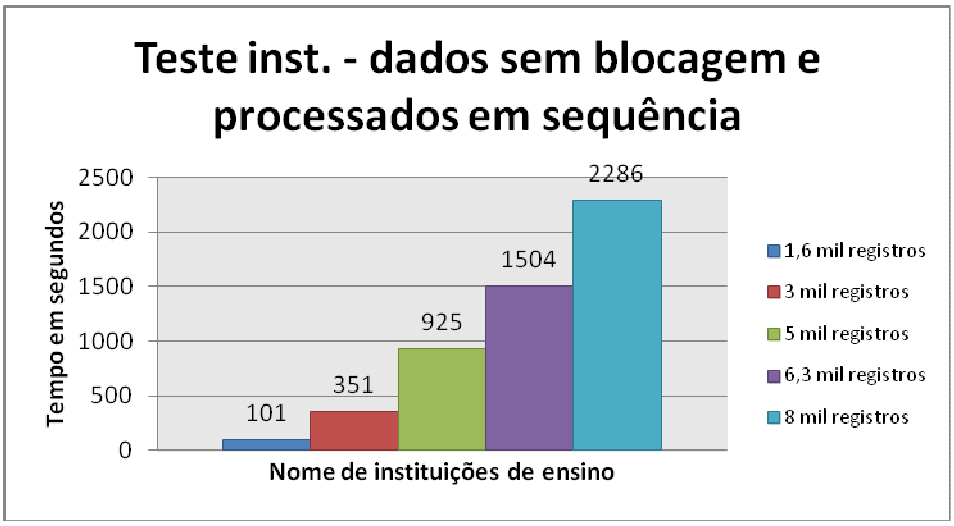


Figura 17 - Teste em nomes de instituições de ensino – tempo de execução para dados sem blocagem e sem paralelização da execução

Para o teste utilizando registros com nome de cidade, **teste cid.**, o tempo de execução aumentou sem seguir um critério especifico. Na execução com as maiores amostras de dados, 7,5 mil e 10 mil, foi onde ocorreu a maior variação de tempo. Isso mostra como a execução das funções de similaridade tem seu desempenho afetado em bases de dados maiores.

Para o **teste rua**, o tempo de execução para os testes aumentou consideravelmente em comparação ao **teste cid.** O comportamento aqui observado é bastante semelhante ao do **teste cid.**, tendo como principal diferença o aumento no tempo de execução para amostras de dados com a mesma quantidade. A diferença no arquivo de saída contendo as comparações que tiveram um escore aceitável também foi semelhante. Então essa diferença de tempo pode ser explicada pelo fato de os registros com nomes de rua serem mais complexos de terem suas similaridades avaliadas.

No **teste inst.**, a amostragem total possuía um número de registros um pouco menor, porém, isso não fez com que a execução dos testes fosse feita em tempo menor. Em comparação com o **teste rua**, o tempo de execução teve como aumento mínimo o dobro do tempo. Já em comparação com o **teste cid.** o aumento foi extremamente maior, algo em torno de 10x mais lento. Avaliando a saída do arquivo, foi verificado que a quantidade de registros que tiveram um score aceitável foi menos que a metade obtida nas avaliações anteriores. Isso indica que, ao serem avaliados *Strings* maiores, dependendo do domínio dos dados, seja necessário reavaliar o limiar de similaridade aceitável.

4.5.2. Dados não blocados utilizando programação paralela

A lógica utilizada nesse teste é a mesma que a do teste anterior, porém foram utilizadas as estruturas de dados paralelas implementadas no Scala, como *ParMap e Parallel Collections.* Para assegurar que os recursos disponíveis, no caso núcleos do processador, estavam sendo completamente utilizados durante o período de testes, a execução foi acompanhada utilizando o “Gerenciador de tarefas” do Windows. Com ele, foi possível visualizar os quatro núcleos do processador sendo usados em 100% durante todo o processo de comparação, enquanto à memória alocada não sofria variações.

Para essa etapa dos testes, os resultados obtidos são apresentados em gráficos conforme a Figura 18 para nomes de cidade, a Figura 19 para nomes de rua e a Figura 20 para instituições de ensino.

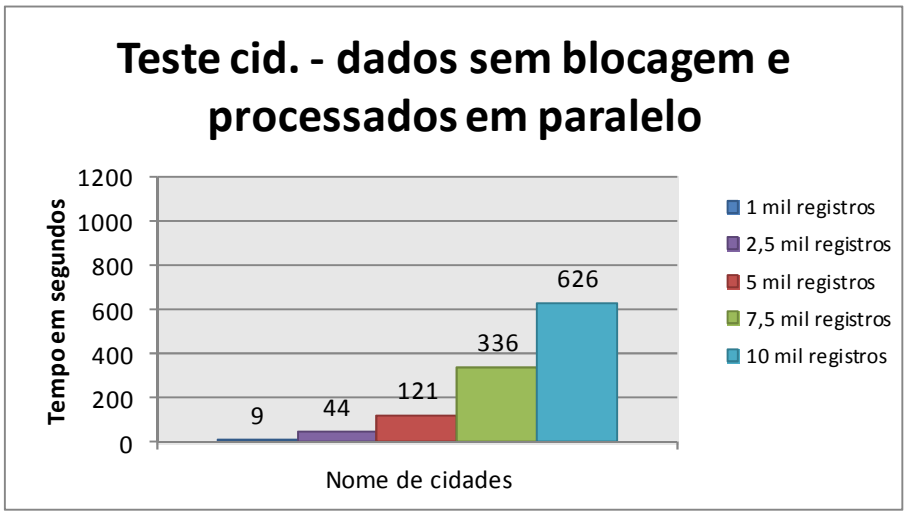


Figura 18 – Teste em nomes de cidade – tempo de execução para dados sem blocagem com paralelização da execução

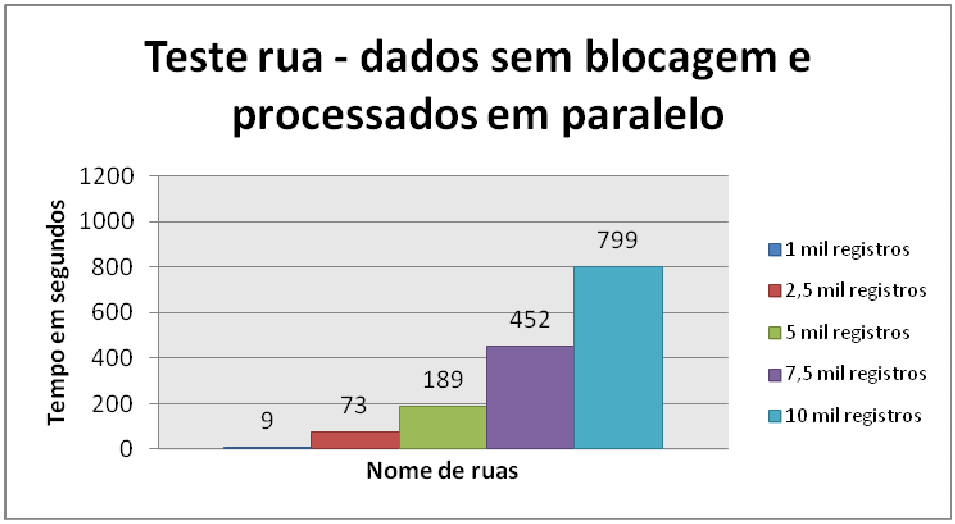


Figura 19 - Teste em nomes de rua – tempo de execução para dados sem blocagem com paralelização da execução

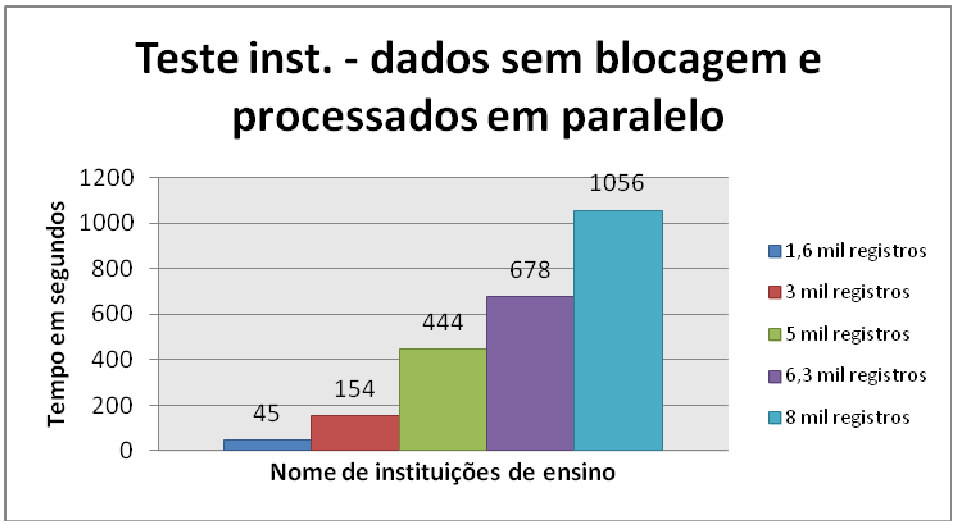


Figura 20 - Teste em nomes de instituições de ensino – tempo de execução para dados sem blocagem com paralelização da execução

Para o **teste cid.,** mesmo tendo o processamento das funções de similaridade em paralelo, o tempo de execução continuou tendo uma maior variação nos últimos grupos de dados onde a quantidade de registros era maior, porém em comparação com o processamento sequencial (Figura 15) a diminuição no tempo de execução variou entre 50% e 100%.

No **teste rua**, os resultados de performance, comparados com a etapa anterior, onde o processamento era sequencial (Figura 16), mostra quão útil é o processamento paralelo para resolução de problemas mais complexos com grande volume de dados. Comparado com o **teste cid.** (Figura 18), os resultados foram semelhantes para amostras de dados menores, como por exemplo, no teste com 1 mil registros, o tempo de execução também foi de 9 segundos. Já para amostras maiores, houve uma diferença de tempo considerável, onde a diferença entre o **teste cid.** e o **teste rua** foi de 173 segundos.

A etapa do **teste inst.** avaliou o comportamento da execução das funções de similaridade em processos paralelos no modelo de produto cartesianos para uma base de dados onde seus registros, no caso *Strings*, possuíam em sua maioria um número de caracteres maior que os avaliados nos outros testes. O comportamento para essa etapa do **teste inst.** seguiu o padrão da mesma etapa para os outros testes, apresentando melhora de desempenho considerável em relação à execução sequencial.

4.5.3. Dados blocados utilizando programação paralela

Para a avaliação da execução de funções de similaridade em dados blocados, o P-BSim trabalhou com dois tipos de índices para os dados: o *Bigram Index* e o *Standart Index.* Nas seções anteriores, foi explicado o funcionamento de cada tipo de indexação. Aqui são apresentados os resultados obtidos para dados que foram previamente blocados utilizando o módulo de blocagem implementado no P-BSim.

Para um melhor aproveitamento do processamento paralelo, antes da execução dos algoritmos de comparação, era carregado um *cache* com os dados blocados em uma estrutura de dados paralela implementada pelo *Scala,* o *ParMap*. Dessa forma, os acessos aos dados eram feitos sem necessidade de *I/O* e em uma estrutura que, por possuir implementações paralelas nativas, conseguia escalonar os acessos, mesmo que simultâneos, as mesmas chaves da melhor forma, não prejudicando o desempenho dos testes.

Para uma melhor organização, os resultados são apresentados separados por tipo de índice usado e explicitando o tipo de dado que foi utilizado durante os experimentos.

- **Bigram Index**

Os testes de comparação realizados em dados que foram previamente blocados utilizando a função *Bigram Index* foram concluídos em tempo bem inferior à comparação sequencial.

No **teste cid.**, o tempo de execução não ultrapassou os 10 segundos em nenhuma das execuções. O tempo de execução aumentava de forma linear em todos os casos, o que indica que a divisão das tarefas realizadas pelas *Parallel Collections* foi feita da mesma maneira independente do

volume de dados avaliado. A única divergência foi na execução do teste utilizando a amostragem com 7,5 mil registros. Mesmo possuindo um número maior de registros a serem avaliados, o seu tempo foi menor que o teste realizado com 5 mil registros. No gráfico apresentado na Figura 21 é possível observar o tempo de execução para cada um dos volumes de dados.

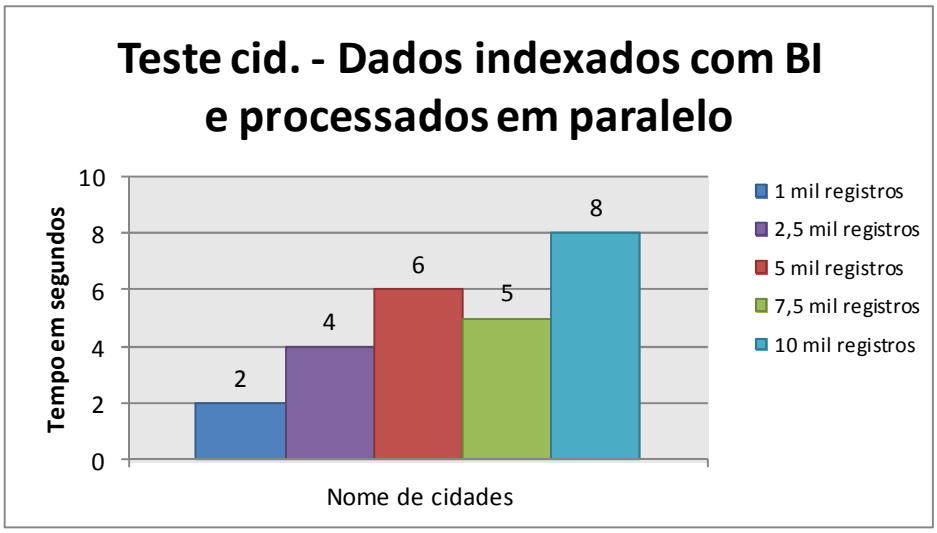


Figura 21 - Teste em nomes de cidade – tempo de execução para dados utilizando BI com paralelização da execução

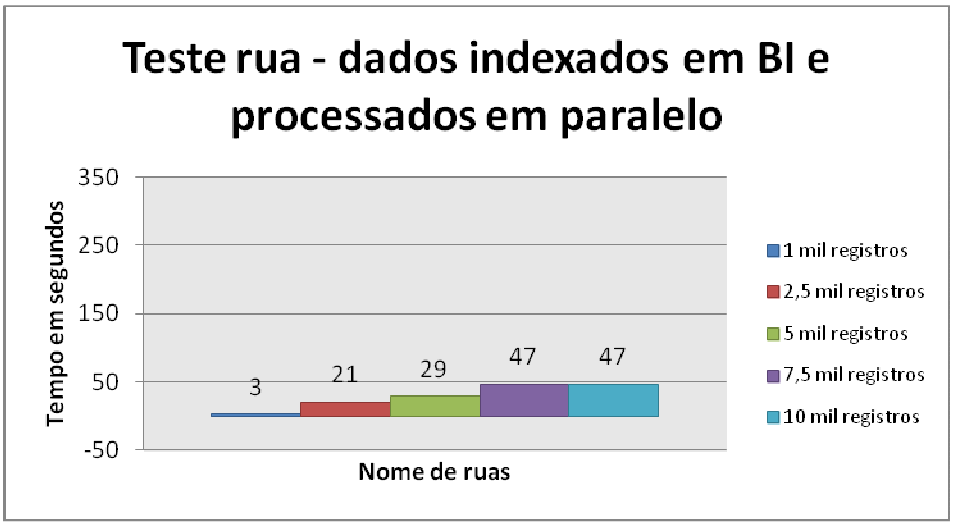


Figura 22 - Teste em nomes de rua – tempo de execução para dados utilizando BI com paralelização da execução

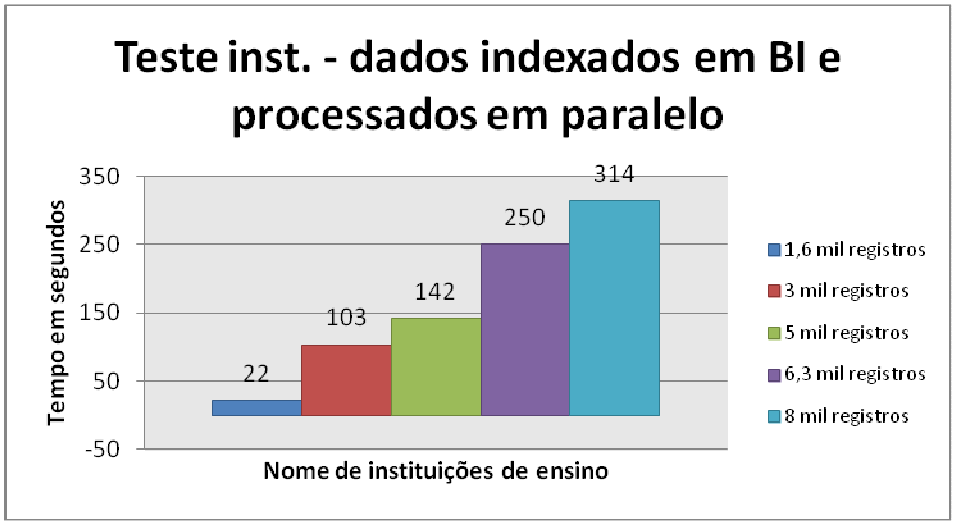


Figura 23 - Teste em nomes de instituições de ensino – tempo de execução para dados utilizando BI com paralelização da execução

Para essa etapa no **teste rua.** (Figura 22), os resultados tiveram a sua maior variação entre a primeira e a segunda amostra de dados, e em contrapartida, a menor variação ocorreu nas duas últimas amostras de dados, onde o tempo de execução foi o mesmo.

O fato de que, para as duas últimas amostras de dados, o tempo de execução foi igual pode ser explicado pelo fato de que o próprio compilador Scala, através da implementação das *Parallel Collections* realiza a divisão das tarefas para o processamento em paralelo, e para o caso da avaliação da amostra contendo 7,5 mil registros, esse escalonamento não foi tão eficiente. Porém, mesmo com essa "falha", os resultados para o processamento em paralelo foram muito melhores se comparados com o processamento sequencial.

O **teste inst.** apresentou o comportamento mais regular, onde a variação de tempo foi aumentando para cada grupo de dados avaliado. Diferente dos outros, não ocorreu nenhum momento onde o tempo de processamento foi igual ou menor para um maior volume de dados. Isso indica que a implementação paralela realizada trabalha melhor para dados que necessitem de um tempo maior de processamento. Assim a divisão de tarefas é feita de forma mais uniforme para todos os casos.

- **Standart Index**

Os testes realizados em dados que foram blocados utilizando a função *Standart Indexing* tiveram resultados bem semelhantes aos testes feitos com dados blocados utilizando *Bigram Indexing* sendo, em alguns casos, até mais rápida. Como na implementação do P-BSim, a chave utilizada na estratégia ST *(Standart Blocking)* é o primeiro caractere de cada palavra do registro, a quantidade de chaves geradas é bem menor. Dessa forma, são feitas mais comparações entre os dados, porém menos iterações na lista de chave.

No **teste cid.**, a execução das comparações também ficou abaixo dos 10 segundos, a quantidade de chaves teve como valor máximo 24. Dessa forma, o número de comparações desnecessário, onde o escore resultante não atinge o mínimo definido foi bem maior. Mesmo assim não houve problemas de desempenho, já que o paralelismo tem sua eficiência aumentada quanto maior a necessidade de computação. No gráfico da Figura 24, é possível visualizar os resultados de maneira mais pontual.

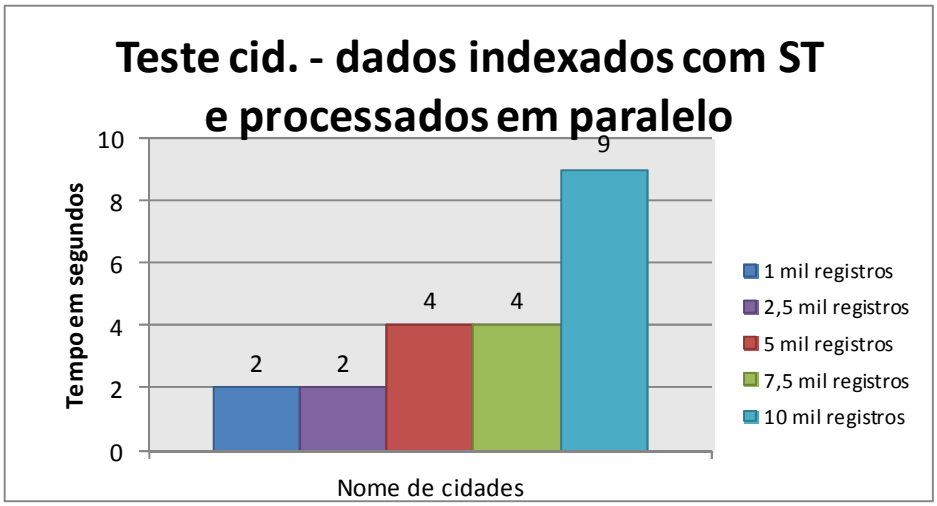


Figura 24 - Teste em nomes de cidade – tempo de execução para dados utilizando ST com paralelização da execução

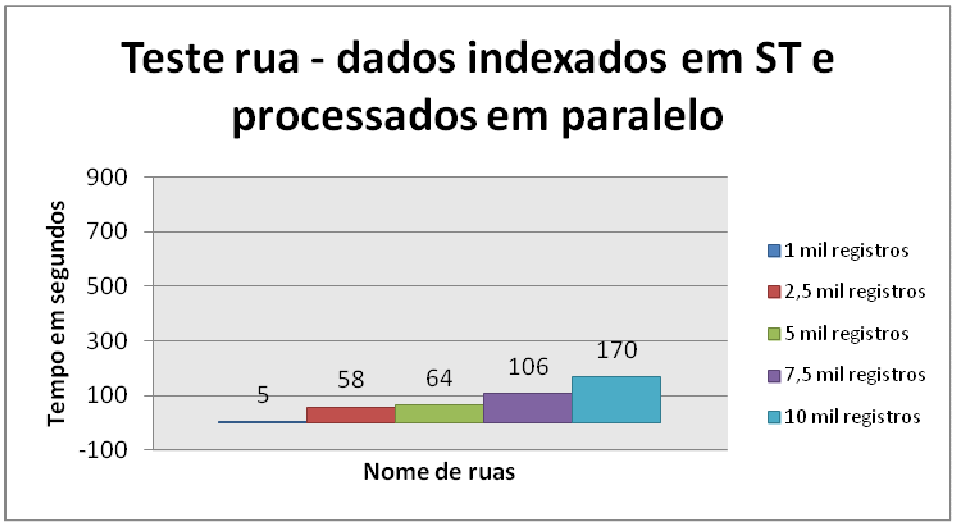


Figura 25 - Teste em nomes de rua – tempo de execução para dados utilizando ST com paralelização da execução

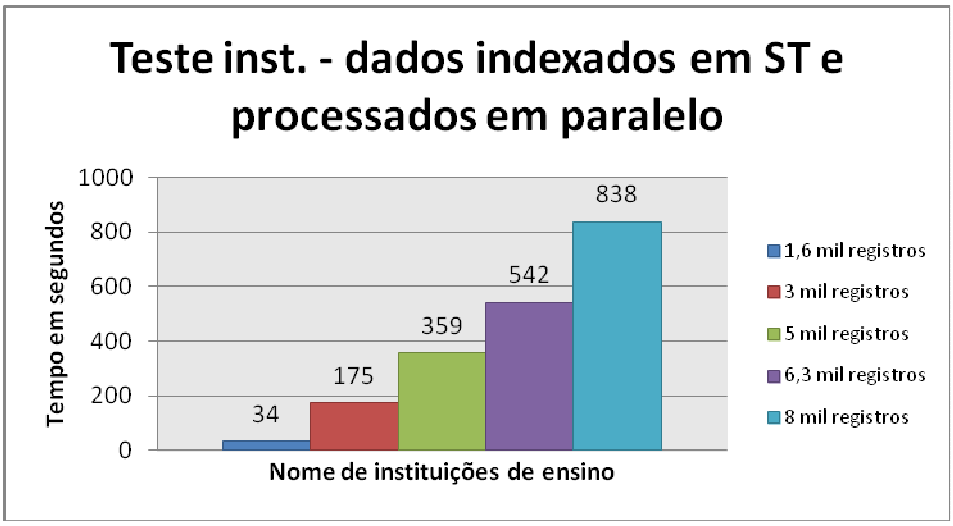


Figura 26 - Teste em nomes de instituições de ensino – tempo de execução para dados utilizando ST com paralelização da execução

No **teste cid.** (Figura 24) nota-se que em dois casos o tempo de execução foi o mesmo, ou seja, mais um indício que a distribuição paralela realizada pelas *Parallel Collections* não é ótima.

Para o **teste rua** é possível perceber que para a execução para volumes de dados variando entre 2,5 e 1,5 mil registros o tempo de execução foi semelhante. No entanto nos extremos, a diferença fica muito mais evidente. A variação total de tempo aqui também foi muito grande, onde o menor valor é 5 segundos e o maior 170 segundos.

O **teste inst.** manteve a regularidade na variação de tempo de execução também nessa etapa. Os resultados podem ser conferidos no gráfico da Figura 26.

A variação do tempo para o **teste inst.** é bastante grande, onde para o menor volume de dados a execução das funções de similaridade leva apenas 34 segundos. Já para o maior volume, esse valor sobe para 383 segundos. Isso demonstra que, por mais regular que seja a variação do tempo em relação à quantidade de registros ela não chega nem próxima de ser uma variação linear.

4.5.4. Dados blocados sem a utilização de programação paralela

Os testes apresentados nessa seção foram realizados utilizando a mesma lógica dos testes apresentados na seção anterior: tanto a questão de dados blocados em um *cache* quanto as regras e *loops* foram exatamente iguais. A mudança principal foi a remoção das estruturas paralelas, substituindo as *Parallels Collections* por *Lists(Java)* ou *Buffers(Scala)* e o *ParMap* por um simples *Map(Java)*. Essa possibilidade de estar utilizando tipos de estruturas das duas linguagens em um mesmo programa é uma característica do Scala, conforme comentado na seção 2.3.

A execução dos testes em dados blocados, mesmo que de forma sequencial, teve uma performance melhor do que os testes realizados nos dados sem blocagem até mesmo utilizando programação paralela na execução das funções.

- **Bigram Index**

No **teste cid.**, a diferença de tempo de execução utilizando processamento paralelo variou entre 1 e 3 segundos dependendo do volume de dados que estava sendo avaliado. O ganho de desempenho é menor do que o ganho que foi obtido paralelizando a análise em dados não blocados. Na Figura 27 é mostrado o gráfico com o tempo de execução para cada um dos grupos de dados.

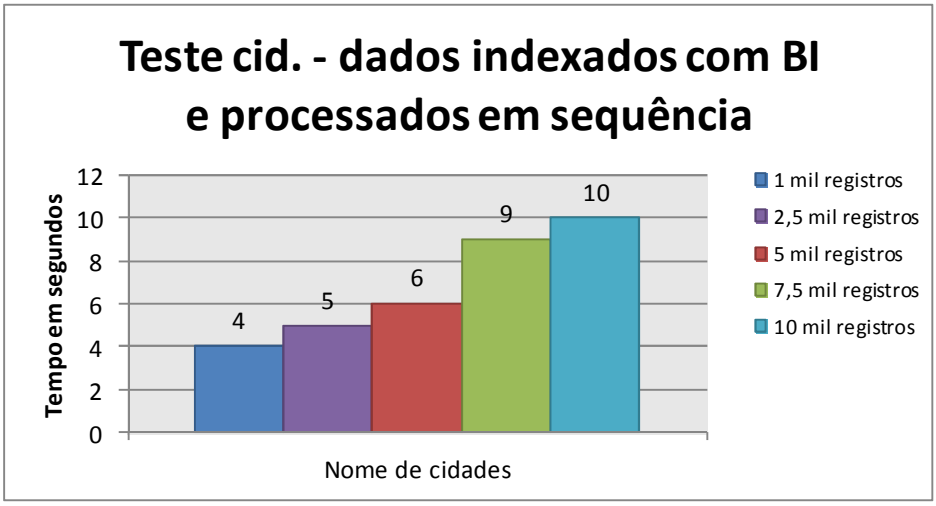


Figura 27 - Teste em nomes de cidade – tempo de execução para dados utilizando BI e processados em sequência

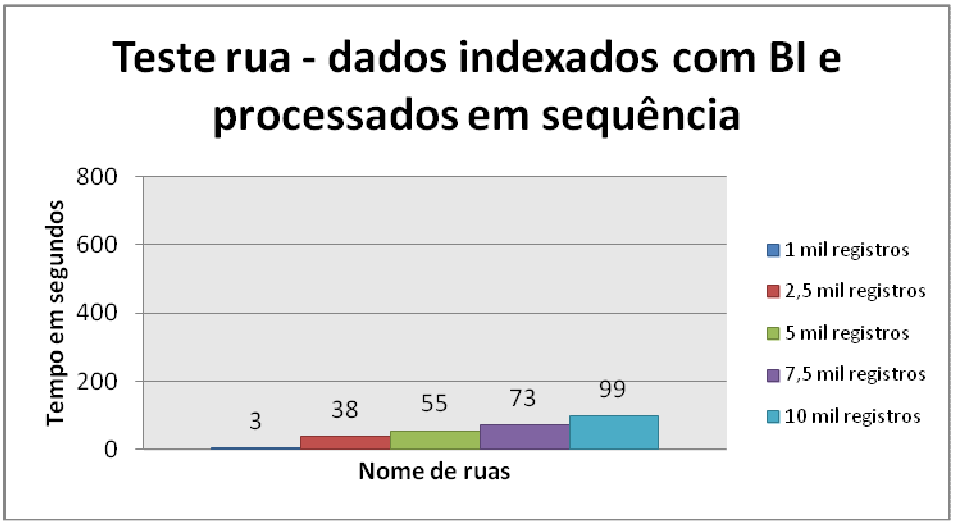


Figura 28 - Teste em nomes de rua – tempo de execução para dados utilizando BI e processados em sequência

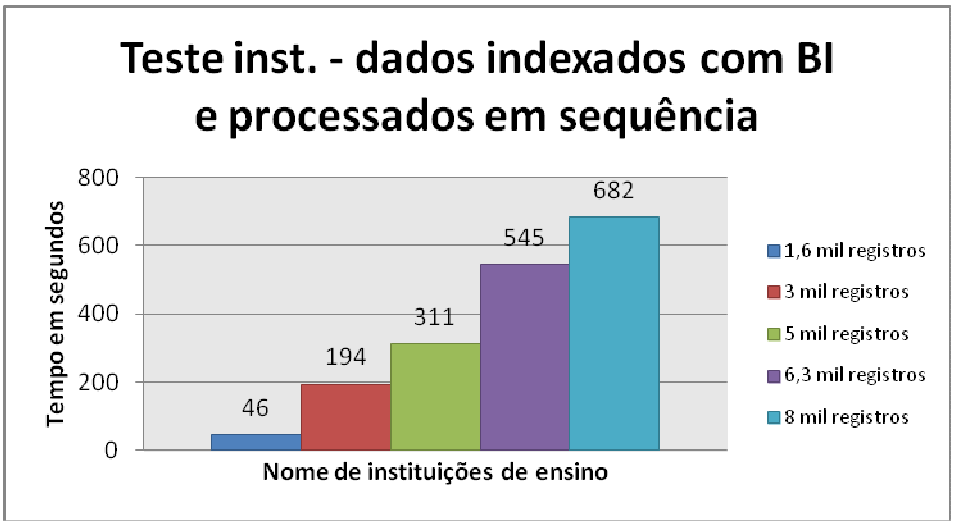


Figura 29 - Teste em nomes de instituições de ensino – tempo de execução para dados utilizando BI e processados em sequência

No **teste rua** os resultados gerados após a execução das funções de similaridade em dados blocados utilizando Bigram Index foram os mostrados na Figura 28.

Nessa etapa do **teste rua**, o comportamento foi semelhante ao observado na execução da etapa anterior, onde o procedimento era o mesmo, porém executado de forma paralela. Até mesmo a maior variação entre o tempo de execução com a primeira e menor amostra (1 mil registros) comparada com a segunda amostra (2,5 mil registros). Um outro ponto bastante interessante foi que a execução sequencial para a primeira amostra levou o mesmo tempo que a execução paralela. Esse fator demonstra que, quanto maior a necessidade de processamento computacional, mais ganho a programação paralela fornece.

A execução desta etapa para o **teste inst.** nos retornou os resultados apresentados na Figura 29. Para o **teste inst.**, o ganho de aproximadamente 100% em relação a performance foi constante indiferentemente à amostra de dados analisada. Para a execução das funções de similaridade em dados blocados utilizando *Bigram Index,* a regularidade entre tamanho da amostra e tempo de execução mostrada nas outros etapas foi mantida.

- **Standart Index**

Para o **teste cid.**, realizando as comparações de forma sequencial em dados que estavam blocados utilizando a indexação ST, os resultados obtidos são apresentados no gráfico da Figura 30.

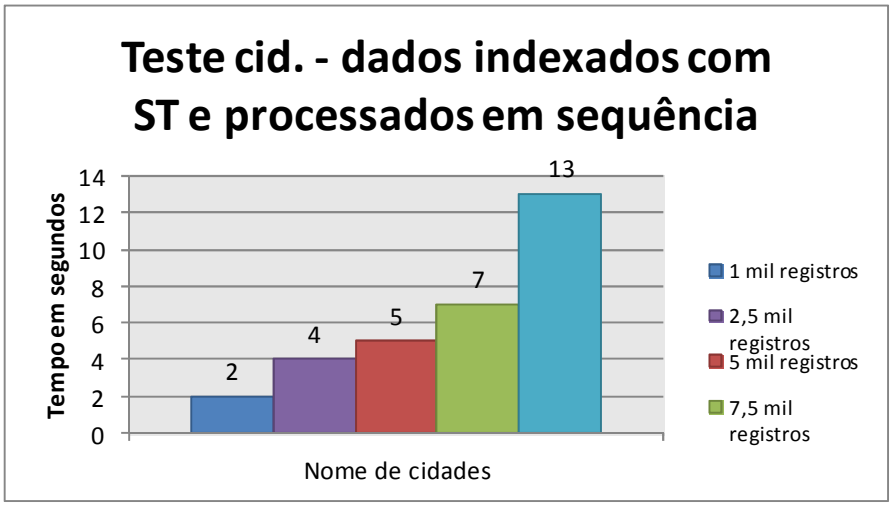


Figura 30 - Teste em nomes de cidade – tempo de execução para dados utilizando ST e processados em sequência

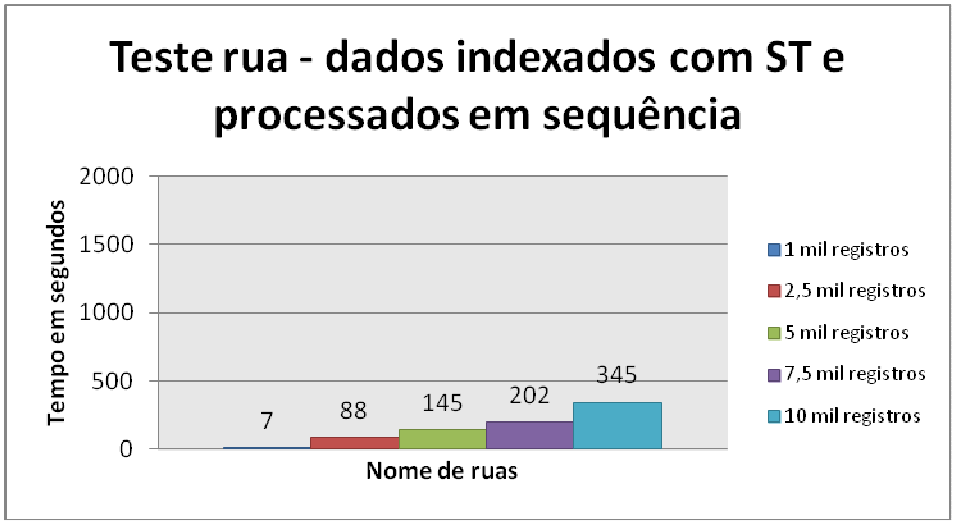


Figura 31 - Teste em nomes de rua – tempo de execução para dados utilizando ST e processados em sequência

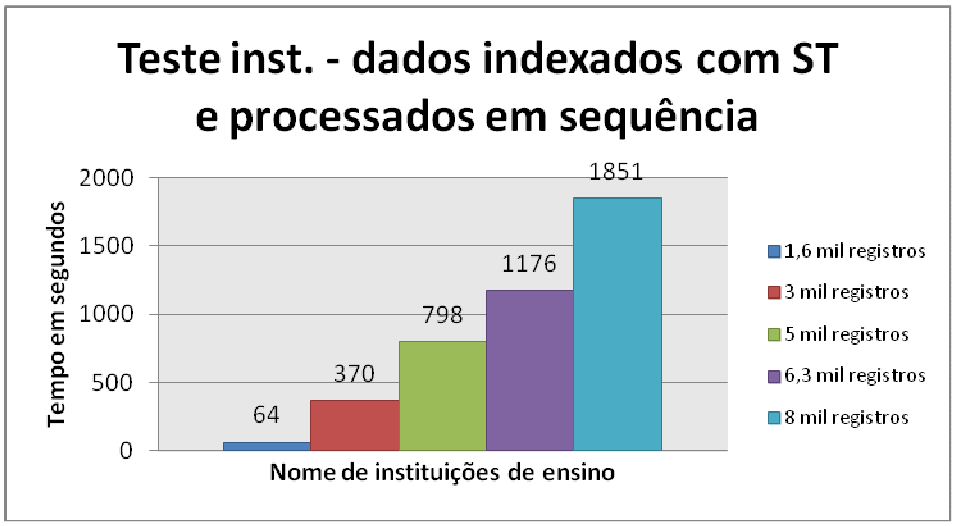


Figura 32 - Teste em nomes de instituições de ensino – tempo de execução para dados utilizando ST e processados em sequência

Ainda para o **teste cid.**, em comparação com a execução paralela do mesmo algoritmo (Figura 24), os ganhos foram mais notórios a partir do grupo com 2,5 mil registros tendo um ganho que variou entre 2 e 4 segundos. Se for realizada uma comparação com a execução do algoritmo trabalhando com dados blocados com *Bigram Index,* percebemos que para os grupos com menor número de registros os resultados obtidos foram melhores.

Para o processamento sequencial das funções de similaridade utilizando dados referentes a nomes de rua (**teste rua**) blocados utilizando o *Standart Index* os resultados retornados são os apresentados na Figura 31. Essa etapa do **teste rua** apresentou um comportamento bastante semelhante à etapa anterior (Figura 25), onde as funções foram executadas de forma paralela. Além do ganho de performance entre a abordagem seqüencial e paralela, comum entre os dois experimentos, nota-se que a distribuição de esforço é bem definida, onde para a primeira amostra (1 mil registros) o tempo de exeçuão de 7 segundos é bastante pequeno se comparado com o da ultima amostra (10 mil registros), que levou 345 segundos para concluir todo o processamento.

A ultima etapa do **teste inst.**, dentro dos testes utilizando dados blocados, foi a mais demorada. O gráfico com os resultados apresentados ao final da execução das funções de similaridade é apresentado na Figura 32. Como em todas as etapas do **teste inst.**, o aumento do tempo de execução ocorreu com certa regularidade. Em geral, essa etapa não apresentou nenhum comportamento diferente dos já registrados nas situações anteriores.

4.5.5. Comparativo entre os resultados

Com os resultados devidamente registrados e catalogados, foi possível realizar algumas análises em cima de cada tipo de teste, observando além de tempo de execução, a qualidade de execução, ou seja, se a saída resultante possuiu excesso de dados redundantes ou se realmente foram feitas somente as comparações necessárias. O comparativo é separado respeitando o número do teste conforme apresentado anteriormente a fim de tornar mais clara a apresentação. Para relembrar, os testes realizados foram:

- **Teste cid.**: Base de dados contendo nome de cidades com 10180 registros
- **Teste rua:** Base de dados contendo nome de ruas com 9895 registros
- **Teste inst.:** Base de dados contendo nome de instituições de ensino contendo 7832 registros.

Para o **teste cid.**, foi possível ver que a blocagem de dados realmente tem um efeito positivo, nesse caso, independente da forma de blocagem escolhida. O gráfico exibido na Figura 33 mostra a evolução do tempo de execução à medida que o volume de dados aumenta para cada um dos experimentos.

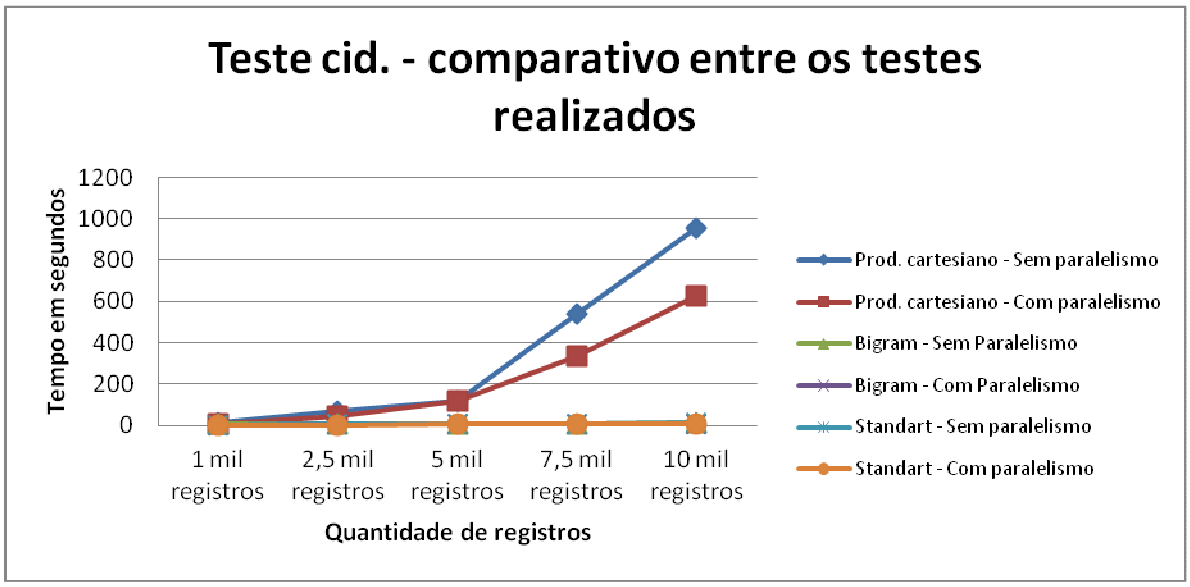


Figura 33 - Teste em nome de cidades - comparativo entre o tempo de execução para a execução de cada teste

A diferença no tempo de execução entre os dados que sofreram blocagem e a comparação sequencial registro a registro (produto cartesiano). O esforço para processar toda a lista de informação é muito maior. Um fato interessante de se observar ainda tratando da comparação no modelo de produto cartesiano (PC) é que, para volumes de dados maiores, o ganho com o paralelismo é maior. Para o **teste cid.** houve uma forte semelhança na execução dos testes com os dados blocados. Por conta disso, foi gerado um novo gráfico somente para os casos que utilizaram dados blocados, o qual é mostrado na Figura 34.

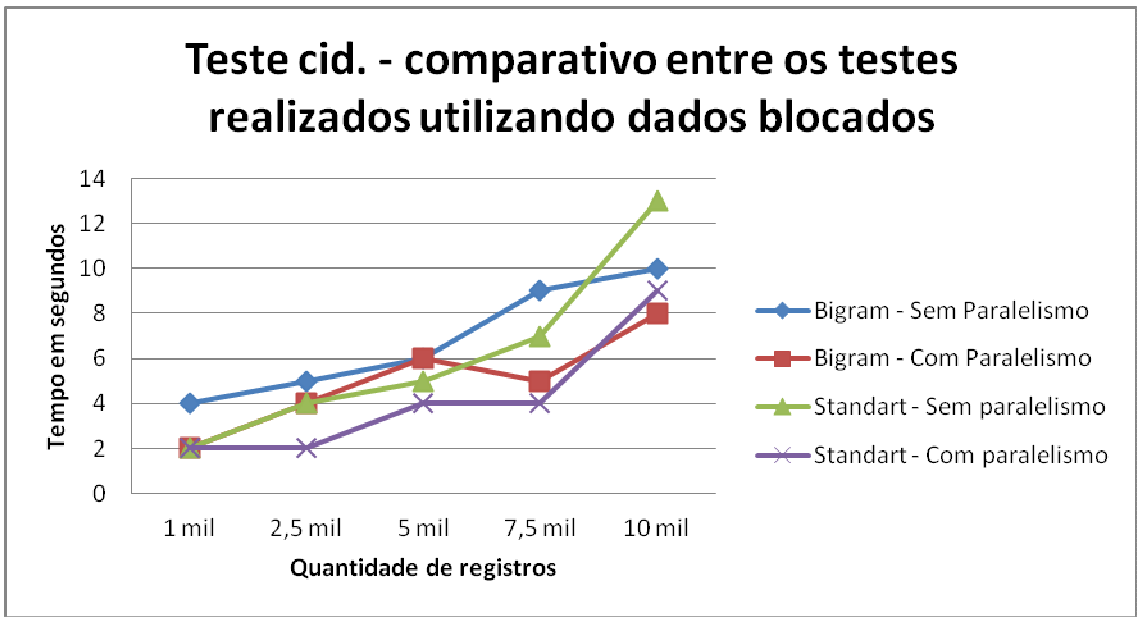


Figura 34 - Teste em nome de cidades - comparativo entre o tempo de execução somente para os testes que utilizaram dados blocados

Quando utilizados dados previamente blocados nas comparações, o tempo de execução não passou dos 14 segundos conforme mostrado no gráfico. Podemos perceber também que não existiu um modelo de blocagem melhor que o outro tendo em vista que os dois modelos alternaram situações onde se saíram melhores, porém a variação apresentada nos testes executados em dados que haviam sido blocados com *Bigram Index* foi menor. Quanto aos benefícios causados pela paralelização do algoritmo, como no restante dos testes, quanto maior o volume de dados que está sendo avaliado, maior a diferença entre o tempo de execução paralela e sequencial.

Quanto à qualidade da saída resultante do algoritmo e o arquivo texto contendo as palavras que respeitaram o limiar de score mínimo para todas as funções, a diferença entre o modelo PC e os blocados foi grande. Quando executada a comparação PC, o arquivo de log contendo as comparações que tiveram escore resultante de acordo com o limiar indicado ficou bastante grande. Por exemplo, caso a base possuísse 300 registros idênticos, os 300 entram no arquivo juntamente com os outros registros que apresentaram escore dentro do padrão, mesmo que não exista relação alguma entre os registros. Já com dados blocados, seguindo com o exemplo dos 300 registros idênticos, na blocagem ST é gerado um número de chaves igual ao numero de palavras. Na BG, o número de chaves é muito variado, mas o aumento da probabilidade de pares relacionados e a eliminação de comparações desnecessárias reduz bastante a quantidade de entradas no arquivo de LOG, tendo em vista que a comparação é feita de forma mais criteriosa, respeitando as chaves geradas na blocagem. As Figuras 35 e 36 mostram dois exemplos de arquivo de saída: um para dados blocados utilizando *Bigram Index* e outro para produto cartesiano (PC).

```
Valores = TELÊMACO BORBA + TELÊMACO BORBA Levenshtein Score = 1.0
Valores = TELÊMACO BORBA + TELÊMACO BORBA BlockDistance Score = 1.0
Valores = TELÊMACO BORBA + TELÊMACO BORBA EuclideanDistance Score = 1.0
Valores = TELÊMACO BORBA + TELÊMACO BORBA Jaro Score = 1.0
Valores = TELÊMACO BORBA + TELÊMACO BORBA Levenshtein Score = 1.0
Valores = TELÊMACO BORBA + TELÊMACO BORBA BlockDistance Score = 1.0
Valores = TELÊMACO BORBA + TELÊMACO BORBA EuclideanDistance Score = 1.0
Valores = TELÊMACO BORBA + TELÊMACO BORBA Jaro Score = 1.0
Valores = TELÊMACO BORBA + TELÊMACO BORBA Levenshtein Score = 1.0
Valores = TELÊMACO BORBA + TELÊMACO BORBA BlockDistance Score = 1.0
Valores = TELÊMACO BORBA + TELÊMACO BORBA EuclideanDistance Score = 1.0
Valores = TELÊMACO BORBA + TELÊMACO BORBA Jaro Score = 1.0
Valores = SANTA IZABEL DO OESTE + SANTA  IZABEL DO OESTE Levenshtein Score = 0.95454544
Valores = SANTA IZABEL DO OESTE + SANTA  IZABEL DO OESTE BlockDistance Score = 1.0
Valores = SANTA IZABEL DO OESTE + SANTA  IZABEL DO OESTE EuclideanDistance Score = 1.0
Valores = SANTA IZABEL DO OESTE + SANTA  IZABEL DO OESTE Jaro Score = 0.9054834
Valores = TELÊMACO BORBA + TELÊMACO BORBA Levenshtein Score = 1.0
Valores = TELÊMACO BORBA + TELÊMACO BORBA BlockDistance Score = 1.0
Valores = TELÊMACO BORBA + TELÊMACO BORBA EuclideanDistance Score = 1.0
```

Figura 35 - Parte do arquivo de LOG resultante do processo de comparação utilizando dados não blocados

```
Valores = SÃO MIGUEL DO OESTE + SÃO MIGUEL DO OESTE BlockDistance Score = 1.0
Valores = SÃO MIGUEL DO OESTE + SÃO MIGUEL DO OESTE EuclideanDistance Score = 1.0
Valores = SÃO MIGUEL DO OESTE + SÃO MIGUEL DO OESTE Jaro Score = 1.0
Valores = SÃO JOSÉ DOS PINHAIS + SÃO JOSÉ DOS PINHAIS Levenshtein Score = 1.0
Valores = SÃO JOSÉ DOS PINHAIS + SÃO JOSÉ DOS PINHAIS BlockDistance Score = 1.0
Valores = SÃO JOSÉ DOS PINHAIS + SÃO JOSÉ DOS PINHAIS EuclideanDistance Score = 1.0
Valores = SÃO JOSÉ DOS PINHAIS + SÃO JOSÉ DOS PINHAIS Jaro Score = 1.0
Valores = SÃO MIGUEL DO OESTE + SÃO MIGUEL DO OESTE Levenshtein Score = 1.0
Valores = SÃO MIGUEL DO OESTE + SÃO MIGUEL DO OESTE BlockDistance Score = 1.0
Valores = SÃO MIGUEL DO OESTE + SÃO MIGUEL DO OESTE EuclideanDistance Score = 1.0
Valores = SÃO MIGUEL DO OESTE + SÃO MIGUEL DO OESTE Jaro Score = 1.0
Valores = SÃO JOSÉ DOS PINHAIS + SÃO JOSÉ DOS PINHAIS Levenshtein Score = 1.0
Valores = SÃO JOSÉ DOS PINHAIS + SÃO JOSÉ DOS PINHAIS BlockDistance Score = 1.0
Valores = SÃO JOSÉ DOS PINHAIS + SÃO JOSÉ DOS PINHAIS EuclideanDistance Score = 1.0
Valores = SÃO JOSÉ DOS PINHAIS + SÃO JOSÉ DOS PINHAIS Jaro Score = 1.0
Valores = SANTO ANDRE + SANTO ANDRE Levenshtein Score = 1.0
Valores = SANTO ANDRE + SANTO ANDRE BlockDistance Score = 1.0
Valores = SANTO ANDRE + SANTO ANDRE EuclideanDistance Score = 1.0
Valores = SANTO ANDRE + SANTO ANDRE Jaro Score = 1.0
Valores = SERRANÓPOLIS DO IOGUAÇU + SERRANÓPOLIS DO IOGUAÇU Levenshtein Score = 1.0
Valores = SERRANÓPOLIS DO IOGUAÇU + SERRANÓPOLIS DO IOGUAÇU BlockDistance Score = 1.0
Valores = SERRANÓPOLIS DO IOGUAÇU + SERRANÓPOLIS DO IOGUAÇU EuclideanDistance Score = 1.0
Valores = SERRANÓPOLIS DO IOGUAÇU + SERRANÓPOLIS DO IOGUAÇU Jaro Score = 1.0
Valores = SANTO ANDRE + SANTO ANDRE Levenshtein Score = 1.0
```

Figura 36 - Parte do arquivo de LOG resultante do processo de comparação utilizando dados blocados com BI

Após uma comparação realizada sobre os testes executados durante o **teste rua** foi montado o gráfico mostrado na Figura 37.

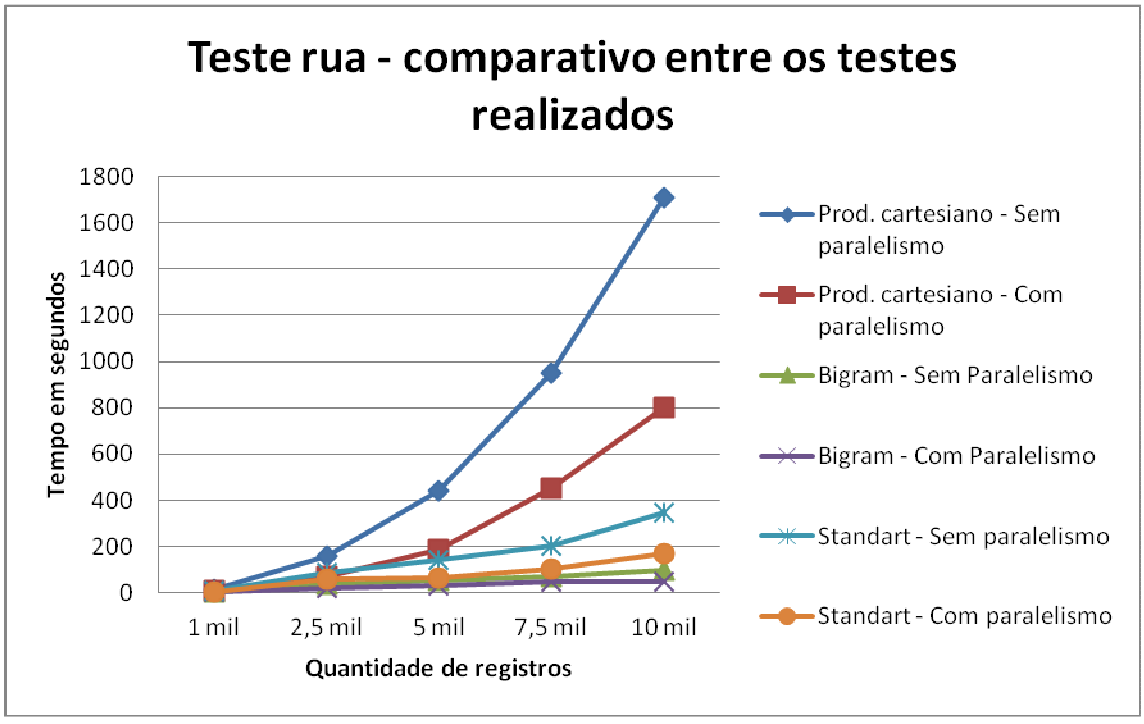


Figura 37 - Teste em nome de ruas - comparativo entre o tempo de execução para a execução de cada teste

Os resultados aqui já ficam um pouco mais esparsos do que os demonstrados no **teste cid.** Através do gráfico percebe-se que os ganhos proporcionados pelo paralelismo são bastante consideráveis considerando o cenário em que o teste foi executado. A comparação feita da forma de produto cartesiano (PC) continua apresentando o maior ganho quanto ao paralelismo já que a variação

entre o tempo da execução paralela e sequencial é sempre menor do que para dados blocados, porém em comparação as execução utilizando dados blocados, nem mesmo a implementação paralela da comparação PC chegou perto de ser tão eficiente.

Aqui também podemos agrupar os experimentos utilizando dados blocados em um outro grupo, devido a tamanha diferença entre a execução em PC e utilizando dados blocados. Porém, os dados blocados com *Bigram Index* apresentaram vantagem sobre os utilizando *Standart Index.* Em alguns casos, mais precisamente para as menores amostras de dados, a diferença foi mínima. Porém a eficiência do Bigram Index não foi superada nem mesmo pela execução paralela das funções de similaridade sob dados blocados com *Standart Index.*

Quanto a qualidade do arquivo de LOG resultante dos testes, o **teste rua** seguiu o padrão apresentado no **teste cid.**

O **teste 3** foi o que apresentou maior equilíbrio nos resultados finais. O ganho com paralelismo foi bastante significativo para as baterias utilizando dados blocados com *Bigram Index* e a execução em modelo produto cartesiano (PC). Com base em todos os resultados, foi gerado o gráfico representado na Figura 38.

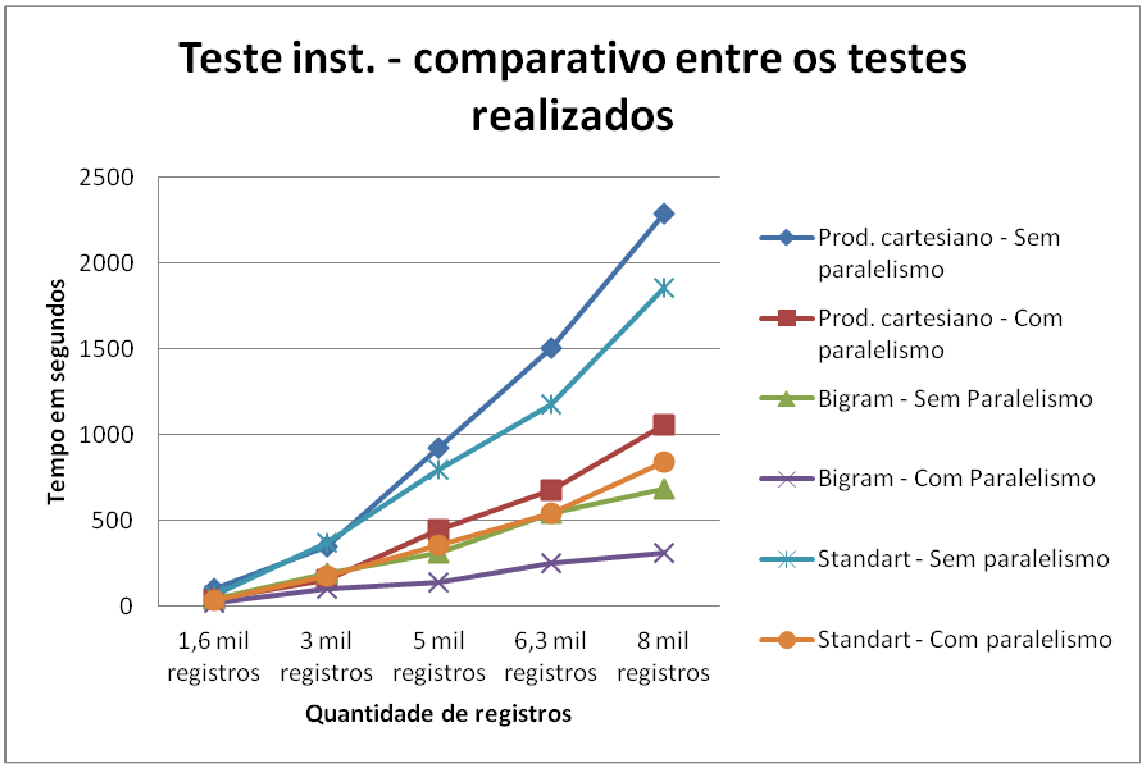


Figura 38 - Teste em nome de instituições de ensino - comparativo entre o tempo de execução para a execução de cada teste

O gráfico mostra que, para os testes utilizando PC e dados blocados com ST, a diferença entre a execução paralela e sequencial segue um padrão, fazendo com a distância entre as retas não seja muito grande. Já nos testes feitos com dados blocados utilizando BI, a medida que o volume de dados aumenta, a distância entre as retas da execução paralela e sequencial ia aumentando de forma mais notória. Isso indica que, dentre todos os testes realizados, a paralelização que mais surtiu efeitos

positivos foi para o teste 3 utilizando dados blocados com BI. A explicação para esse fato é que, para *Strings* com várias palavras, como é o caso dos nomes de instituições de ensino, a blocagem utilizando BI acaba gerando um número de chaves grande. Para as amostras com mais registros, o número de chaves acaba sendo maior que a quantidade de registros da tabela original. Dessa forma, a demanda por processamento é maior, e é nesse ponto que o paralelismo trabalha melhor. Quanto mais processamento necessário, mais beneficio em performance a paralelização apresenta.

Para uma melhor visualização entre as diferenças apresentadas em cada tipo de teste, foi gerado um gráfico tendo como padrão a execução dos testes em uma amostragem de 5 mil registros. O gráfico comparativo entre os três testes realizados é representado na Figura 39.

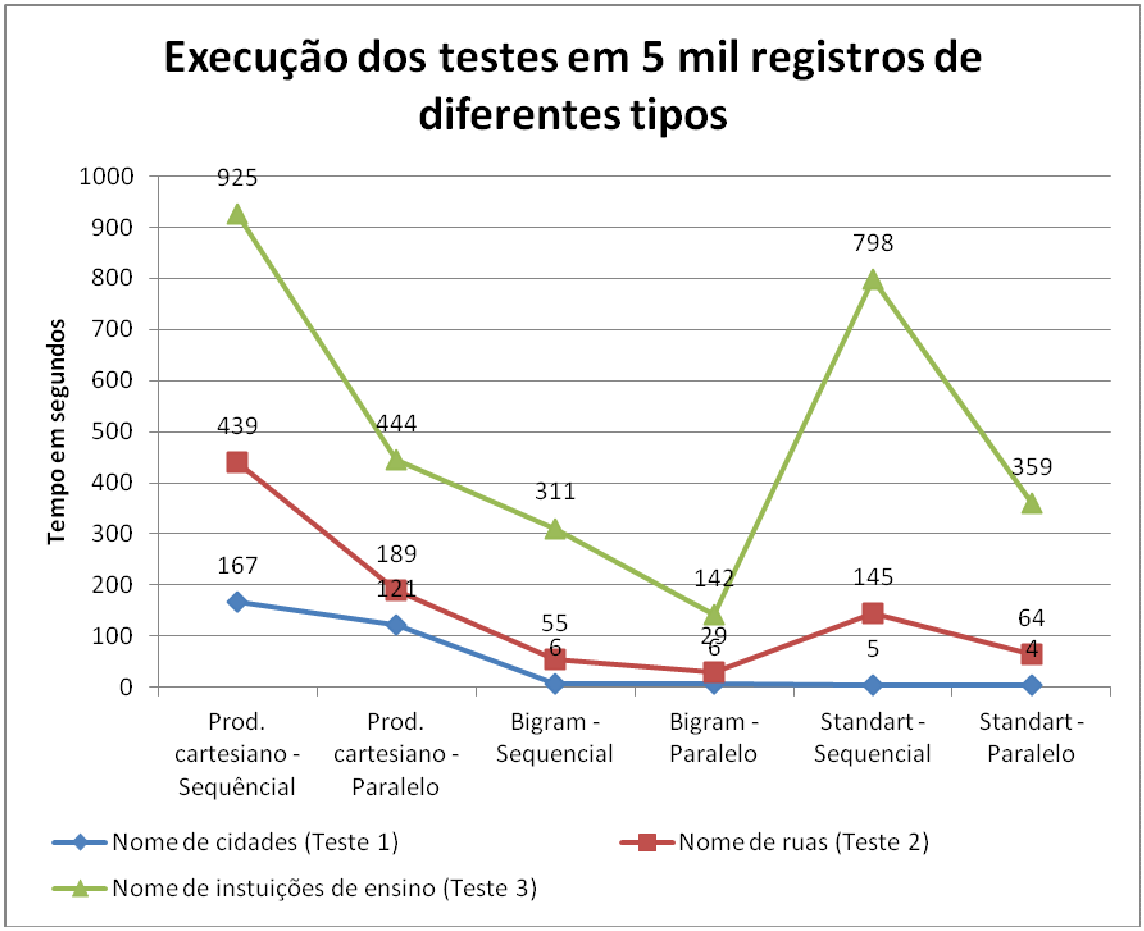


Figura 39 - Comparativo entre os testes executados para uma mesma quantidade de registros na tabela original

Para uma quantidade de dados iguais, no caso 5 mil registros, podemos ver que o **teste inst.** foi o que mais levou tempo para processar todas as funções de similaridade. Isso indica que, conforme citado anteriormente, quanto maior a *String* que está sendo avaliada, mais demorada é a execução das funções de similaridade. O **teste cid. e rua** apresentaram características bastante semelhantes, tanto em variação no tempo de execução quanto nos ganhos utilizando a programação paralela. Já o teste 3 é o

que apresenta o gráfico mais “desnivelado” demonstrando que foi o teste que mais teve ganhos quando utilizando o processamento paralelo.

Os maiores ganhos na execução dos algoritmos em paralelo foram registrados nas etapas de teste utilizando produto cartesiano e blocagem utilizando *Standart Index*. Isso comprova que, conforme citado acima, o paralelismo de algoritmos apresenta melhores resultados quando utilizado em um cenário que exija grande processamento computacional, ou seja, em execuções de muitas tarefas. Como a blocagem utilizando *Bigram Index* se apresentou mais eficiente quanto à geração de pares candidatos mais prováveis, o processamento desnecessário com pares candidatos não semelhantes não é realizado.

5. Conclusão e Trabalhos Futuros

Nesse trabalho, foram apresentados alguns resultados que a programação paralela pode trazer para a resolução do problema de deduplicação. Foram mostradas também algumas técnicas de blocagem, o seu funcionamento e a sua implementação. Os resultados demonstrados ao final do trabalho deixam claro o benefício que uma programação paralela pode trazer quanto ao desempenho da execução de funções de similaridade. A utilização ou não de alguma técnica de blocagem de dados antes de execução das funções também se mostrou favorável à utilização de uma blocagem, mesmo que simples. O tipo de blocagem é algo que deve ser escolhido com enorme critério, pois dependendo do domínio do problema a blocagem acaba por prejudicar a performance da execução das funções de similaridade. Por exemplo, a blocagem de registros contendo nomes de artigos pode não apresentar grandes ganhos, tendo em vista que o registro é composto normalmente de muitas palavras, o que acarreta em um numero de chaves provavelmente bem superior a quantidade de registros da tabela original. Outro fator que se apresentou relevante quanto à execução de funções de similaridade é o tamanho das *Strings* a serem avaliadas, o que torna o conhecimento do domínio do problema bastante interessante para saber qual a melhor estratégia a ser utilizada.

Com uma diferença média no tempo de execução variando entre 75 e 100% nas hipóteses positivas, as funções de similaridade utilizando a estrutura de *Parallel Collections e ParMap* fornecida pela linguagem *Scala*, demonstrou que é possível realizar processamento concorrente de forma mais simples, porém não menos eficaz. Os resultados obtidos foram em todos os casos positivos, tornando a relação entre custo e beneficio de um desenvolvimento mais elaborado para o mesmo problema muito bom. Em questões arquiteturais foi a abordagem mais simples encontrada para se trabalhar com esse tipo de problema.

Em questão de desempenho focado em cada um dos três domínios de dados utilizados nesse trabalho, é possível afirmar que: para o domínio de dados utilizando nomes de cidades, a abordagem que ofereceu melhores resultados de modo geral foi a que utilizou dados blocados com *Standart Índex* e processados em paralelo. Para o domínio de dados utilizando nomes de ruas, a melhor abordagem no geral foi a que utilizou dados blocados com *Bigram Índex* e processados em paralelo. O ultimo domínio de dados, nomes de instituições de ensino, obteve um melhor desempenho quando utilizados dados blocados com *Bigram Índex* e processados em paralelo.

Por fim, esse trabalho visou esclarecer, corroborando com a literatura, a necessidade de que cada vez mais as aplicações sejam concebidas com o conceito de paralelismo, pois em grande parte do uso, o processador permanece com núcleos ociosos enquanto o desempenho do sistema que está se utilizando não é das melhores. Ainda foi mostrado que a utilização de estruturas paralelas, ou *"programação multi-thread"*, vem sendo simplificada permitindo assim a utilização desses conceitos em diversos cenários.

Como trabalhos a serem desenvolvidos no mesmo foco que o P-BSim estão:

- Implementar a blocagem de dados utilizando o mesmo conceito de programação paralela utilizada para a execução das funções de similaridade.
- Buscar formas mais eficientes, porém não mais complexas de executar funções de similaridade entre registros ou qualquer outro processo necessário para solucionar o problema de deduplicação de dados.
- Aplicar o conceito de programação paralela aqui apresentado em problemas computacionais mais complexos, saindo da vertente de banco de dados.
- Realizar experimentos com bases de dados de outros domínios e com um volume da dados maior.
- Avaliar as métricas abordadas em [Gonçalvez, 2008]. Completitude dos Pares (CP) e a Taxa de Redução (TR), além da *F-score* (FS) e a Qualidade dos Pares (QP).
- Tentar solucionar o problema na performance em execuções de funções de similaridade em *Strings* contendo muitos caracteres.
- Submissão de artigo para conferência, relatando o trabalho e o resultado dos experimentos.

6. Referências

Santos, W.; Teixeira, T.; Machado, C.; Meira, Jr. W.; S. Da Silva, A; Ferreira,R.; Guedes, G. **A Scalable Parallel Deduplication Algorithm**, 2007, UFMG

Kawai, H.; Garcia-Molina, H.; Benjelloun, O.; Menestrina, D.; Whang, E.; Gong, H. **P-Swoosh Parallel Algorithm For Generic Entity Resolution**, 2008, Stanford

Friedrich Dorneles, C.; De Matos Galante, R. **Aplicação De Funções De Similaridade E Detecção De Diferenças Em Grandes Volumes De Dados Distribuídos,** 2008, PUC-Rio

Rohan Baxter; Peter Christen; Tim Churches **A Comparison of Fast Blocking Methods for Record Linkage,** 2003, Australian National University

C. F. Dorneles, C. A. Heuser, V. M. Orengo, A. S. da Silva, E. S. de Moura: **A strategy for allowing meaningful and comparable scores in approximate matching. Information System, vol 34, n 08**, December 2009.

Guilherme Dal Bianco, Renata Galante, Mirella Moura Moro: **P-Canopy: uma proposta para blocagem paralela.** In: IX Escola Regional de Alto Desempenho - Arquiteturas Multicore (ERAD), March 2009, Caxias do Sul, RS, Brazil

Martin Odersky, Lex Spoon, and Bill Venners. **Programming In Scala First Edition, Version 6;** 2008

Walter dos Santos Filho: **Algoritmo paralelo e eficiente para o problema de pareamento de dados,** 2008, UFMG, Belo Horizonte, MG, Brasil.

Bilenko,M., Kamath, B., andMooney, R. J. **Adaptive blocking: Learning to scale up record linkage,** 2006. In ICDM '06: Proceedings of the Sixth International Conference on Data Mining, pages 87–96, Washington, DC, USA. IEEE Computer Society.

Christen, P. **Performance and scalability of fast blocking techniques for deduplication and data linkage**, 2007, In VLDB 2007: 33rd International Conference on Very Large Data Bases, Vienna, Austria. ACM.

F. Gonçalves, C.; Santos, W.; F. D. Flores, L.; S. Vilela, M.; Machado, C.; Meira Jr., W.; Silva, A. **Avaliação De Técnicas Paralelas De Blocagem Para Resolução De Entidades E Deduplicação,** 2008, UFMG

Hernandez, M. A. and Stolfo, S. J. **Real-world data is dirty: Data cleansing and the merge/purge problem**, 1998, Data Mining and Knowledge Discovery, 2(1):9–37.

Aleksandar Prokopec, Tiark Rompf, Phil Bagwell, Martin Odersky **A Generic Parallel Collection Framework,** 2010

Subramaniam, V. **Programming Scala** Tackle Multi-Core Complexity On The Java Virtual Machine. Isbn: 978-1-93435-631-9

Febrl - Freely extensible biomedical record linkage. Disponível em: http://cs.anu.edu.au/~Peter.Christen/Febrl/febrl-0.3/febrldoc-0.3/node33.html

Api Scala. Disponível Em: Http://Www.Scala-Lang.Org/Api/Current/Index.Html

7. Apêndice 1 – Código fonte

```
package br.com.pSim.blocking.ngram;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
/**
 *
 * @author Gustavo de Geus
 *
 */
public class Ngram {

	private String palavra;
	private int granularidade;

	  public Ngram(int granularidade) {
			this.granularidade = granularidade;
		}

	  /**
	   * Método que percorre uma String, extrai todos os "N"grams possiveis e os ordena
	   *
	   * @return Lista com os todos os "N"grams possiveis para a palavra em ordem
	   * alfabetica
	   */
	  public List<String> getOrdenedGrams (String str) {
		  this.palavra = this.formataString(str);
	    List<String> grams = new ArrayList<String>();
	    String bg;
	    int limiteLoop = this.palavra.length() - (granularidade - 1);
	    for (int i = 0; i < limiteLoop; i++) {
	      bg = this.palavra.substring(i, i+granularidade);
	      grams.add(bg);
	    }
	    Collections.sort(grams);
	    return grams;
```

```
	}

	private String formataString(String str) {
		String palavraFormatada = str.replaceAll(" ", "");
		palavraFormatada = palavraFormatada.toUpperCase();
			return palavraFormatada;
		}

	/**
	 * Main teste
	 *
	 * @param args
	 */
	public static void main(String[] args)
	{
		//BIGRAM
	   Ngram bg = new Ngram(2);
	   for (String string : bg.getOrdenedGrams("CATOLICA"))
	   {
	      System.out.println(string);
	   }
//	   //TRIGRAM
//	   Ngram tg = new Ngram(3);
//	   for (String string : tg.getOrdenedGrams("peter"))
//	   {
//	      System.out.println(string);
//	   }

	}

}

package br.com.pSim.blocking.ngram;

import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

```
import br.com.pSim.utils.CombinationGenerator;
import br.com.pSimUtils.bd.dao.BigramBlockingDAO;
import br.com.pSimUtils.bd.dao.INGramDAO;
import br.com.pSimUtils.bd.dao.TrigramBlockingDAO;

public class NgramBlocking {

	private List<String> elementos;
	private Double threshould;
	private int granularidade;
	private INGramDAO ngramDao;
	private String currentItem;

	public NgramBlocking(List<String> palavras, Double threshould,
			int granularidade, String tableName) {
		this.elementos = palavras;
		this.threshould = threshould;
		this.granularidade = granularidade;
		this.getDaoInstance(granularidade);
		this.ngramDao.setTableName(tableName);
	}

	private void getDaoInstance(int granularidade) {
		if (granularidade == 2) {
			ngramDao = new BigramBlockingDAO();
		} else {
			if (granularidade == 3) {
				ngramDao = new TrigramBlockingDAO();
			}
		}

	}

	public void generateBlocks() {
		List<String> chaves;
		String[] palavras;
		String fraseFormatada;
```

```
                for (String frase : elementos) {
                        fraseFormatada = this.formataFrase(frase);
                        palavras = fraseFormatada.split(" ");
                        for (String palavra : palavras) {
                                if (palavra != null && palavra.length() > granularidade) {
                                        chaves = this.getKeys(palavra);
                                        for (String key : chaves) {
                                                if (key != null) {
                                                        if (this.containsKey(key)) {

        ngramDao.atualizaEntradaTabela(key, currentItem
                                                                                        +
fraseFormatada);
                                                        } else {

        ngramDao.insereEntradaTabela(key,

        fraseFormatada);
                                                        }
                                                }
                                        } else {
                                                if (this.containsKey(palavra)) {

                                                        ngramDao.atualizaEntradaTabela(palavra,
currentItem
                                                                        + fraseFormatada);
                                                } else {
                                                        ngramDao.insereEntradaTabela(palavra,
fraseFormatada);
                                                }
                                        }
                                }
                        }
                }

                protected String formataFrase(String frase) {
                        String padrao = "\\s{2,}";
```

```
		Pattern regPat = Pattern.compile(padrao);
		if (frase != null) {
			Matcher matcher = regPat.matcher(frase);
			String res = matcher.replaceAll(" ").trim();
			return res;
		}
		System.out.println("Frase nula " + frase);
		return "";
	}

	private boolean containsKey(String key) {
		currentItem = ngramDao.getValue(key);
		if (currentItem != null && !currentItem.equals("")) {
			return true;
		}
		return false;
	}

	public List<String> getKeys(String palavra) {
		List<String> keys = new ArrayList<String>();
		if (palavra != null && !palavra.isEmpty() && palavra.length() > 2) {
			Ngram ng = new Ngram(granularidade);
			List<String> grams = ng.getOrdenedGrams(palavra);
			CombinationGenerator cmbGen = new CombinationGenerator(
					grams.size(),
this.getCombinationSize(palavra.length()));
			keys = cmbGen.getCombinations(grams);
		} else {
			keys.add(palavra);
		}
		return keys;
	}

	private int getCombinationSize(int size) {
		Double aux = Double.valueOf(size);
		Double result = aux * threshould;
		return result.intValue();
	}
```

```
	public static void main(String[] args) {
		List<String> palavras = new ArrayList<String>();
		palavras.add("ZILAH DOS SANTOS BATISTA");
		NgramBlocking ngb = new NgramBlocking(palavras, 0.9, 2, "filmes");
		// FEDERALDESANTAMARIA
		List<String> keys = ngb.getKeys("CASA");
		for (String string : keys) {
			System.out.println(string);
		}

	}

}

package br.com.pSim.blocking.SB;

import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import br.com.pSimUtils.bd.dao.StandartBlockingDAO;

public class StandartBlocking {

	private List<String> elementos;
	private String currentItem;
	private String nomeTabela;
	private StandartBlockingDAO standartBlockingDAO;

	public StandartBlocking(List<String> elementos, String nomeTable) {
		this.elementos = elementos;
		this.nomeTabela = nomeTable;
		standartBlockingDAO = new StandartBlockingDAO();
	}

	/**
	 * Metodo que separa em blocos as palavras que começarem com a mesma letra.
```

```
	 *
	 * @return
	 */
	public void generateBlocks() {
		Character primeiraLetra;
		String[] palavras;
		standartBlockingDAO.setTableName(nomeTabela);
		String fraseFormatada;
		for (String frase : elementos) {
			fraseFormatada = this.formataFrase(frase);
			palavras = fraseFormatada.split(" ");
			for (String palavra : palavras) {
				primeiraLetra = palavra.toUpperCase().charAt(0);
				if (this.contaisKey(primeiraLetra)) {

	standartBlockingDAO.atualizaEntradaTabela(

	String.valueOf(primeiraLetra), currentItem
                                                                                    +
fraseFormatada);
				} else {
					standartBlockingDAO.insereEntradaTabela(
                                                String.valueOf(primeiraLetra),
fraseFormatada);
				}
			}
		}
	}

	private String formataFrase(String frase) {
		String padrao = "\\s{2,}";
		Pattern regPat = Pattern.compile(padrao);
		if (frase != null) {
			Matcher matcher = regPat.matcher(frase);
			String res = matcher.replaceAll(" ").trim();
			return res;
		}
		System.out.println("Frase nula " + frase);
```

```
                return "";
        }

        private boolean contaisKey(Character primeiraLetra) {
                currentItem = standartBlockingDAO.getValue(String
                                .valueOf(primeiraLetra));
                if (currentItem != null && !currentItem.equals("")) {
                        return true;
                }
                return false;
        }

}

package br.com.pSim.test;

import java.util.Date;
import java.util.List;

import br.com.pSim.blocking.SB.StandartBlocking;
import br.com.pSim.blocking.ngram.NgramBlocking;
import br.com.pSimUtils.bd.dao.DataCollectorDAO;

public class BlockingGeneratorTest {

        public void generateNGramBlocks (String tableName, int cardinalidade, List<String>
dados) {
                Long dtInicio = new Date().getTime();
                List<String> data = dados;
                System.out.println("Generate **Bigram Blocks** started with " + data.size() +
" elements");
                NgramBlocking ngBlock = new NgramBlocking(data, 0.9, cardinalidade,
tableName);
                ngBlock.generateBlocks();
                System.out.println("Generate Blocks finished!!!");
                Long tempoTotal = new Date().getTime() - dtInicio;
                System.out.println("Tempo total --> " + tempoTotal/1000 + " seg.");
        }
```

```
		public void generateStandartBlocks (String tableName, List<String> dados) {
			Long dtInicio = new Date().getTime();
			List<String> data = dados;
			System.out.println("Generate **Standart Blocks** started with " + data.size()
+ " elements");
			StandartBlocking stdBlock = new StandartBlocking(data, tableName);
			stdBlock.generateBlocks();
			System.out.println("Generate Blocks finished!!!");
			Long tempoTotal = new Date().getTime() - dtInicio;
			System.out.println("Tempo total --> " + tempoTotal/1000 + " seg.");
		}

		public static void main(String[] args) {
			BlockingGeneratorTest blockTeste = new BlockingGeneratorTest();
	//		List<String> artigos = DataCollectorDAO.getArtigos();
	//		List<String> cidades = DataCollectorDAO.getCidades();
	//		List<String> concursos = DataCollectorDAO.getConcursos();
			List<String> instituicoes = DataCollectorDAO.getInstituicoes();
	//		List<String> ruas  = DataCollectorDAO.getRuas();
	//		blockTeste.generateNGramBlocks("artigos", 2, artigos);
	//		blockTeste.generateStandartBlocks("artigos", artigos);
	//		blockTeste.generateStandartBlocks("cidades", cidades);
	//		blockTeste.generateNGramBlocks("cidades", 2, cidades);
	//		blockTeste.generateStandartBlocks("concursos", concursos);
	//		blockTeste.generateNGramBlocks("concursos", 2, concursos);
			blockTeste.generateStandartBlocks("instituicoes", instituicoes);
			blockTeste.generateNGramBlocks("instituicoes", 2, instituicoes);
	//		blockTeste.generateNGramBlocks("ruas", 2, ruas);
	//		blockTeste.generateStandartBlocks("ruas", ruas);
			System.exit(0);
		}

	}

	package br.com.pSim.utils;
```

```
/**
 * Classe recebe um array da string e gera todas as combinacoes possiveis
 * entre os elementros desse array do tamanho que for estabelecido.
 *
 * @author Gustavo de Geus
 * @version $Revision: 1.2 $
 */
//----------------------------------------
//Systematically generate combinations.
//----------------------------------------

import java.math.BigInteger;
import java.util.ArrayList;
import java.util.List;

public class CombinationGenerator
{

   private int[] a;

   private int n;

   private int r;

   private BigInteger numLeft;

   private BigInteger total;

   //------------
   // Constructor
   //------------

   public CombinationGenerator(int n, int r)
   {
      if (r > n)
      {
//          throw new IllegalArgumentException();
            System.out.println("R maior que N - " + this.getClass().getCanonicalName());
```

```
        }
        if (n < 1)
        {
//          throw new IllegalArgumentException();
           System.out.println("N menor que 1 - " + this.getClass().getCanonicalName());
        }
        this.n = n;
        this.r = r;
        a = new int[r];
        BigInteger nFact = getFactorial(n);
        BigInteger rFact = getFactorial(r);
        BigInteger nminusrFact = getFactorial(n - r);
        total = nFact.divide(rFact.multiply(nminusrFact));
        reset();
    }

    //------
    // Reset
    //------

    public void reset()
    {
        for (int i = 0; i < a.length; i++)
        {
            a[i] = i;
        }
        numLeft = new BigInteger(total.toString());
    }

    //------------------------------------------------
    // Return number of combinations not yet generated
    //------------------------------------------------

    public BigInteger getNumLeft()
    {
        return numLeft;
    }
```

```
//-----------------------------
// Are there more combinations?
//-----------------------------

public boolean hasMore()
{
   return numLeft.compareTo(BigInteger.ZERO) == 1;
}

//--------------------------------------
// Return total number of combinations
//--------------------------------------

public BigInteger getTotal()
{
   return total;
}

//------------------
// Compute factorial
//------------------

private static BigInteger getFactorial(int n)
{
   BigInteger fact = BigInteger.ONE;
   for (int i = n; i > 1; i--)
   {
      fact = fact.multiply(new BigInteger(Integer.toString(i)));
   }
   return fact;
}

//-----------------------------------------------------------
// Generate next combination (algorithm from Rosen p. 286)
//-----------------------------------------------------------

public int[] getNext()
{
```

```
    if (numLeft.equals(total))
    {
        numLeft = numLeft.subtract(BigInteger.ONE);
        return a;
    }

    int i = r - 1;
    while (a[i] == n - r + i)
    {
        i--;
    }
    a[i] = a[i] + 1;
    for (int j = i + 1; j < r; j++)
    {
        a[j] = a[i] + j - i;
    }

    numLeft = numLeft.subtract(BigInteger.ONE);
    return a;

}


        public List<String> getCombinations(List<String> palavras) {
                int[] indices;
                StringBuffer combination = null;
                List<String> combinations = new ArrayList<String>();
                while (this.hasMore()) {
                        combination = new StringBuffer();
                        indices = this.getNext();
                        for (int i = 0; i < indices.length; i++) {
                                String item = palavras.get(indices[i]);
                                combination.append(item);
                        }
                        combinations.add(combination.toString());
                        combination = null;
                }
```

```
                return combinations;
        }


        public static void main(String[] args) {
                List<String> palavras = new ArrayList<String>();

                palavras.add("pr");
                palavras.add("ri");
                palavras.add("is");
                palavras.add("sc");
                palavras.add("ci");
                palavras.add("il");
                palavras.add("la");

                CombinationGenerator combinationGen = new
CombinationGenerator(palavras.size(), 6);
                List<String> combination = combinationGen.getCombinations(palavras);
                for (String string : combination) {
                        System.out.println(string);
                }
        }
}


package br.com.pSim.utils;

import java.io.File;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.FileOutputStream;
import java.io.IOException;
import java.util.Date;

public class LogGenerator {

        File file;
        FileOutputStream fileIO;
```

```
public LogGenerator() {
	this.createLogFile();
}



private void createLogFile () {
	Date currentDate = new Date();
	file = new File("pSimExecution-" + currentDate.getTime() + ".log");
	try {
		fileIO = new FileOutputStream(file);
		String header = this.headreGenerate(currentDate, file.getName());
		fileIO.write(header.getBytes());
	} catch (FileNotFoundException fnfe) {
		fnfe.printStackTrace();
	} catch (IOException ioe) {
		ioe.printStackTrace();
	}
}

private String headreGenerate(Date currentDate, String name) {

	StringBuilder sb = new StringBuilder();
	sb.append("****************************");
	sb.append("* pSim Execution LogFile  *");
	sb.append("* Author: Gustavo de Geus *");
	sb.append("* Date:");
	sb.append(currentDate.toString());
	sb.append("*");
	return sb.toString();
}



public void insetLogData (String info) {
	try {
```

```
                        String identedInfo = info.concat("\n");
                        this.fileIO.write(identedInfo.getBytes());

                } catch (IOException ioe) {
                        ioe.printStackTrace();
                }
        }

        public static void main(String[] args) {
            try {
              // Gravando no arquivo
              File arquivo;

              arquivo = new File("arquivo.txt");
              FileOutputStream fos = new FileOutputStream(arquivo);
              String texto = "quero gravar este texto no arquivo";
              fos.write(texto.getBytes());
              texto = "\nquero gravar este texto AQUI TAMBEM";
              fos.write(texto.getBytes());
              fos.close();

              // Lendo do arquivo
              arquivo = new File("arquivo.txt");
              FileInputStream fis = new FileInputStream(arquivo);

              int ln;
              while ( (ln = fis.read()) != -1 ) {
                System.out.print( (char)ln );
              }

              fis.close();
            }
            catch (Exception ee) {
              ee.printStackTrace();
            }
          }
}
```

```
package br.com.pSimUtils.bd.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

import br.com.pSimUtils.db.manager.DBConnector;

public class BigramBlockingDAO implements INGramDAO {

	private String tableName;

	@Override
	public void insereEntradaTabela(String chave, String valor) {
		StringBuilder query = new StringBuilder();
		String consulta;
		PreparedStatement prepStmt = null;
		Connection conect = null;

		try {
			conect = DBConnector.getConnection();

			query.append("INSERT INTO " + tableName + "_BLOCKS_BIGRAM_8K
");
			query.append("(CHAVE, VALORES) VALUES (?,?)");

			consulta = query.toString();

			prepStmt = conect.prepareStatement(consulta);
			prepStmt.setString(1, chave);
			prepStmt.setString(2, valor + "#");
			prepStmt.executeUpdate();
```

```
		} catch (SQLException sqe) {
			sqe.printStackTrace();
		} catch (ClassNotFoundException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		} finally {
			try {
				DBConnector.cleanup(conect, prepStmt);
			} catch (SQLException sqe) {
				sqe.printStackTrace();
			}
		}
	}

	@Override
	public void atualizaEntradaTabela(String chave, String valor) {
		StringBuilder query = new StringBuilder();
		String consulta;
		PreparedStatement prepStmt = null;
		Connection conect = null;

		try {
			conect = DBConnector.getConnection();

			query.append("UPDATE " + tableName + "_BLOCKS_BIGRAM_8K ");
			query.append("SET VALORES = ?");
			query.append("WHERE CHAVE = ?");

			consulta = query.toString();

			prepStmt = conect.prepareStatement(consulta);
			prepStmt.setString(1, valor + "#");
			prepStmt.setString(2, chave);
			prepStmt.executeUpdate();

		} catch (SQLException sqe) {
			sqe.printStackTrace();
		} catch (ClassNotFoundException e) {
```

```
                    // TODO Auto-generated catch block
                    e.printStackTrace();
                }

                finally {
                    try {
                        DBConnector.cleanup(conect, prepStmt);
                    } catch (SQLException sqe) {
                        sqe.printStackTrace();
                    }
                }
            }

            @Override
            public String getValue(String chave) {
                StringBuilder query = new StringBuilder();
                String consulta;
                PreparedStatement prepStmt = null;
                Connection conect = null;
                ResultSet resultSet;
                String value = null;

                try {
                    conect = DBConnector.getConnection();

                    query.append("SELECT VALORES FROM " + tableName +
"_BLOCKS_BIGRAM_8K ");
                    query.append("WHERE CHAVE = ?");

                    consulta = query.toString();

                    prepStmt = conect.prepareStatement(consulta);
                    prepStmt.setString(1, chave);
                    resultSet = prepStmt.executeQuery();

                    while (resultSet.next()) {
                        value = resultSet.getString(1);
                    }
```

```
		} catch (SQLException sqe) {
			sqe.printStackTrace();
		} catch (ClassNotFoundException e) {
			e.printStackTrace();
		}

		finally {
			try {
				DBConnector.cleanup(conect, prepStmt);
			} catch (SQLException sqe) {
				sqe.printStackTrace();
			}
		}
		return value;
	}

	public Map<String, List<String>> loadCache (int quantity)
	{
		HashMap<String, List<String>> result = new HashMap<String, List<String>>();
		StringBuilder query = new StringBuilder();
		String consulta;
		PreparedStatement prepStmt = null;
		Connection conect = null;
		ResultSet resultSet;
		List<String> value = new ArrayList<String>();

		try {
			conect = DBConnector.getConnection();

			query.append("SELECT CHAVE, VALORES FROM " + this.tableName +
"_BLOCKS_BIGRAM_" + quantity +"K ");

			consulta = query.toString();

			prepStmt = conect.prepareStatement(consulta);
			resultSet = prepStmt.executeQuery();
```

```
                while (resultSet.next()) {
                        String chave = resultSet.getString(1);
                        String valueStr = resultSet.getString(2);
                        value = new ArrayList<String>();
                        for (String string : valueStr.split("#")) {
                                value.add(string);
                        }
                        result.put(chave, value);
                }

        } catch (SQLException sqe) {
                sqe.printStackTrace();
        } catch (ClassNotFoundException e) {
                e.printStackTrace();
        }

        finally {
                try {
                        DBConnector.cleanup(conect, prepStmt);
                } catch (SQLException sqe) {
                        sqe.printStackTrace();
                }
        }
        return result;
}


public synchronized List<String> getValueList(String chave) {
        StringBuilder query = new StringBuilder();
        String consulta;
        PreparedStatement prepStmt = null;
        Connection conect = null;
        ResultSet resultSet;
        List<String> value = new ArrayList<String>();

        try {
                conect = DBConnector.getConnection();
```

```
                query.append("SELECT VALORES FROM " + tableName +
"_BLOCKS_BIGRAM_8K ");
                query.append("WHERE CHAVE = ?");

                consulta = query.toString();

                prepStmt = conect.prepareStatement(consulta);
                prepStmt.setString(1, chave);
                resultSet = prepStmt.executeQuery();

                while (resultSet.next()) {
                        String valueStr = resultSet.getString(1);
                        for (String string : valueStr.split("#")) {
                                value.add(string);
                        }
                }

        } catch (SQLException sqe) {
                sqe.printStackTrace();
        } catch (ClassNotFoundException e) {
                e.printStackTrace();
        }

        finally {
                try {
                        DBConnector.cleanup(conect, prepStmt);
                } catch (SQLException sqe) {
                        sqe.printStackTrace();
                }
        }
        return value;
}


public synchronized List<String> getValuesList(List<String> chaves) {
        StringBuilder query = new StringBuilder();
        String consulta;
        PreparedStatement prepStmt = null;
```

```
Connection conect = null;
ResultSet resultSet;
List<String> value = new ArrayList<String>();
List<String> chavesJava = new ArrayList<String>();
chavesJava.addAll(chaves);

try {
	conect = DBConnector.getConnection();

	query.append("SELECT VALORES FROM " + tableName +
"_BLOCKS_BIGRAM_8K ");
	query.append("WHERE CHAVE = ? ");
	for (int j = 1; j < chavesJava.size(); j++) {
		query.append(" OR CHAVE = ?");

	}
	consulta = query.toString();

	prepStmt = conect.prepareStatement(consulta);
	prepStmt.setString(1, chavesJava.get(0));
	for (int i = 1; i<chavesJava.size(); i++) {
		prepStmt.setString(i + 1, chavesJava.get(i));

	}
	resultSet = prepStmt.executeQuery();

	while (resultSet.next()) {
		String valueStr = resultSet.getString(1);
		for (String string : valueStr.split("#")) {
			value.add(string);
		}
	}

} catch (SQLException sqe) {
	sqe.printStackTrace();
} catch (ClassNotFoundException e) {
	e.printStackTrace();
}
```

```
                finally {
                        try {
                                DBConnector.cleanup(conect, prepStmt);
                        } catch (SQLException sqe) {
                                sqe.printStackTrace();
                        }
                }
                return value;
        }

        @Override
        public void setTableName(String tableName) {
                this.tableName = tableName;

        }

        public static void main(String[] args) {
                BigramBlockingDAO blockingDAO = new BigramBlockingDAO();
                List<String> chaves = new ArrayList<String>();
                chaves.add("AXIMXA");
                chaves.add("AXIMXI");
                chaves.add("AXXAXI");
                chaves.add("IMXAXI");
                blockingDAO.setTableName("cidades");
                blockingDAO.getValuesList(chaves);
        }

}

package br.com.pSimUtils.bd.dao;

import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.ArrayList;
import java.util.List;
```

```
import br.com.pSimUtils.db.manager.DBConnector;



public class DataCollectorDAO {

	public static List<String> getCidades() {

		StringBuilder query = new StringBuilder();
		String consulta;
		PreparedStatement prepStmt = null;
		Connection conect = null;
		ResultSet resultSet;
		List<String> cidades = new ArrayList<String>();

		try {
			conect = DBConnector.getConnection();

			query.append("SELECT CIDADE FROM CIDADES ORDER BY CIDADE
DESC LIMIT 1000");

			consulta = query.toString();

			prepStmt = conect.prepareStatement(consulta);
			resultSet = prepStmt.executeQuery();

			while (resultSet.next()) {
				cidades.add(resultSet.getString(1));
			}

		} catch (SQLException sqe) {
			sqe.printStackTrace();
		} catch (ClassNotFoundException e) {
			e.printStackTrace();
		}

		finally {
```

```
                try {
                        DBConnector.closeConnection(conect);
                } catch (SQLException sqe) {
                        sqe.printStackTrace();
                }
        }
        return cidades;
}

public static List<String> getArtigos () {
        StringBuilder query = new StringBuilder();
        String consulta;
        PreparedStatement prepStmt = null;
        Connection conect = null;
        ResultSet resultSet;
        List<String> cidades = new ArrayList<String>();

        try {
                conect = DBConnector.getConnection();

                query.append("SELECT TITLE FROM ARTIGOS ORDER BY TITLE DESC
LIMIT 3000");

                consulta = query.toString();

                prepStmt = conect.prepareStatement(consulta);
                resultSet = prepStmt.executeQuery();

                while (resultSet.next()) {
                        cidades.add(resultSet.getString(1));
                }

        } catch (SQLException sqe) {
                sqe.printStackTrace();
        } catch (ClassNotFoundException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
```

```
		finally {
			try {
				DBConnector.closeConnection(conect);
			} catch (SQLException sqe) {
				sqe.printStackTrace();
			}
		}
		return cidades;
	}

	public static List<String> getRuas () {
		StringBuilder query = new StringBuilder();
		String consulta;
		PreparedStatement prepStmt = null;
		Connection conect = null;
		ResultSet resultSet;
		List<String> cidades = new ArrayList<String>();

		try {
			conect = DBConnector.getConnection();

			query.append("SELECT RUA FROM RUAS ORDER BY RUA DESC");

			consulta = query.toString();

			prepStmt = conect.prepareStatement(consulta);
			resultSet = prepStmt.executeQuery();

			while (resultSet.next()) {
				cidades.add(resultSet.getString(1));
			}

		} catch (SQLException sqe) {
			sqe.printStackTrace();
		} catch (ClassNotFoundException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
```

```
		}

		finally {
			try {
				DBConnector.closeConnection(conect);
			} catch (SQLException sqe) {
				sqe.printStackTrace();
			}
		}
		return cidades;
	}

	public static List<String> getRuasSemDuplicatas () {
		StringBuilder query = new StringBuilder();
		String consulta;
		PreparedStatement prepStmt = null;
		Connection conect = null;
		ResultSet resultSet;
		List<String> cidades = new ArrayList<String>();

		try {
			conect = DBConnector.getConnection();

			query.append("SELECT RUA FROM RUAS_SEM_DUPLICATAS");

			consulta = query.toString();

			prepStmt = conect.prepareStatement(consulta);
			resultSet = prepStmt.executeQuery();

			while (resultSet.next()) {
				cidades.add(resultSet.getString(1));
			}

		} catch (SQLException sqe) {
			sqe.printStackTrace();
		} catch (ClassNotFoundException e) {
			// TODO Auto-generated catch block
```

```
                e.printStackTrace();
        }

        finally {
                try {
                        DBConnector.closeConnection(conect);
                } catch (SQLException sqe) {
                        sqe.printStackTrace();
                }
        }
        return cidades;
}

public static List<String> getFilmes () {
        StringBuilder query = new StringBuilder();
        String consulta;
        PreparedStatement prepStmt = null;
        Connection conect = null;
        ResultSet resultSet;
        List<String> artigos = new ArrayList<String>();

        try {
                conect = DBConnector.getConnection();

                query.append("SELECT TITLE FROM FILMES");

                consulta = query.toString();

                prepStmt = conect.prepareStatement(consulta);
                resultSet = prepStmt.executeQuery();

                while (resultSet.next()) {
                        artigos.add(resultSet.getString(1));
                }

        } catch (SQLException sqe) {
                sqe.printStackTrace();
        } catch (ClassNotFoundException e) {
```

```
                // TODO Auto-generated catch block
                e.printStackTrace();
        }

        finally {
                try {
                        DBConnector.closeConnection(conect);
                } catch (SQLException sqe) {
                        sqe.printStackTrace();
                }
        }
        return artigos;
}

public static List<String> getConcursos() {
        StringBuilder query = new StringBuilder();
        String consulta;
        PreparedStatement prepStmt = null;
        Connection conect = null;
        ResultSet resultSet;
        List<String> enderecos = new ArrayList<String>();

        try {
                conect = DBConnector.getConnection();

                query.append("SELECT ENDER FROM CONCURSOS");

                consulta = query.toString();

                prepStmt = conect.prepareStatement(consulta);
                resultSet = prepStmt.executeQuery();

                while (resultSet.next()) {
                        enderecos.add(resultSet.getString(1));
                }

        } catch (SQLException sqe) {
                sqe.printStackTrace();
```

```
		} catch (ClassNotFoundException e) {
			// TODO Auto-generated catch block
			e.printStackTrace();
		}

		finally {
			try {
				DBConnector.closeConnection(conect);
			} catch (SQLException sqe) {
				sqe.printStackTrace();
			}
		}
		return enderecos;
	}

	public static List<String> getInstituicoes() {
		StringBuilder query = new StringBuilder();
		String consulta;
		PreparedStatement prepStmt = null;
		Connection conect = null;
		ResultSet resultSet;
		List<String> instituicoes = new ArrayList<String>();

		try {
			conect = DBConnector.getConnection();

			query.append("SELECT NOME FROM INSTITUICOES ORDER BY NOME
DESC");

			consulta = query.toString();

			prepStmt = conect.prepareStatement(consulta);
			resultSet = prepStmt.executeQuery();

			while (resultSet.next()) {
				instituicoes.add(resultSet.getString(1));
			}
```

```
                        } catch (SQLException sqe) {
                                sqe.printStackTrace();
                        } catch (ClassNotFoundException e) {
                                // TODO Auto-generated catch block
                                e.printStackTrace();
                        }

                        finally {
                                try {
                                        DBConnector.closeConnection(conect);
                                } catch (SQLException sqe) {
                                        sqe.printStackTrace();
                                }
                        }
                        return instituicoes;
                }
        }
```

```
package br.com.pSimUtils.bd.dao;
public interface INGramDAO {
        public void setTableName(String tableName);
        public void insereEntradaTabela(String chave, String valor);
        public void atualizaEntradaTabela(String chave, String valor);
        public String getValue(String chave);
}
```

```
        package br.com.pSimUtils.bd.dao;

        import java.sql.Connection;
        import java.sql.PreparedStatement;
        import java.sql.ResultSet;
        import java.sql.SQLException;
        import java.util.ArrayList;
        import java.util.HashMap;
        import java.util.List;
        import java.util.Map;
```

```
import br.com.pSimUtils.db.manager.DBConnector;

public class StandartBlockingDAO{

	private String tableName;

	public void insereEntradaTabela(String chave, String valor) {
		StringBuilder query = new StringBuilder();
		String consulta;
		PreparedStatement prepStmt = null;
		Connection conect = null;

		try {
			conect = DBConnector.getConnection();

			query.append("INSERT INTO " + this.tableName +
"_blocks_standart_8K ");
			query.append("(CHAVE, VALORES) VALUES (?,?)");

			consulta = query.toString();

			prepStmt = conect.prepareStatement(consulta);
			prepStmt.setString(1, chave);
			prepStmt.setString(2, valor+ "#");
			prepStmt.executeUpdate();

		} catch (SQLException sqe) {
			sqe.printStackTrace();
		} catch (ClassNotFoundException e) {
			e.printStackTrace();
		}
		finally {
			try {
				DBConnector.cleanup(conect, prepStmt);
			} catch (SQLException sqe) {
				sqe.printStackTrace();
			}
```

```
		}

	}

	public void atualizaEntradaTabela(String chave, String valor) {
		StringBuilder query = new StringBuilder();
		String consulta;
		PreparedStatement prepStmt = null;
		Connection conect = null;

		try {
			conect = DBConnector.getConnection();

			query.append("UPDATE " + this.tableName + "_blocks_standart_8K
");
			query.append("SET VALORES = ? ");
			query.append("WHERE CHAVE = ?");

			consulta = query.toString();

			prepStmt = conect.prepareStatement(consulta);
			prepStmt.setString(1, valor+ "#");
			prepStmt.setString(2, chave);
			prepStmt.executeUpdate();

		} catch (SQLException sqe) {
			sqe.printStackTrace();
		} catch (ClassNotFoundException e) {
			e.printStackTrace();
		}

		finally {
			try {
				DBConnector.cleanup(conect, prepStmt);
			} catch (SQLException sqe) {
				sqe.printStackTrace();
			}
```

```
		}

	}

	public Map<String, List<String>> loadCache (int quantity)
	{
		HashMap<String, List<String>> result = new HashMap<String, List<String>>();
		StringBuilder query = new StringBuilder();
		String consulta;
		PreparedStatement prepStmt = null;
		Connection conect = null;
		ResultSet resultSet;
		List<String> value = new ArrayList<String>();

		try {
			conect = DBConnector.getConnection();

			query.append("SELECT CHAVE, VALORES FROM " + this.tableName +
"_blocks_standart_" + quantity +"K ");

			consulta = query.toString();

			prepStmt = conect.prepareStatement(consulta);
			resultSet = prepStmt.executeQuery();

			while (resultSet.next()) {
				String chave = resultSet.getString(1);
				String valueStr = resultSet.getString(2);
				value = new ArrayList<String>();
				for (String string : valueStr.split("#")) {
					value.add(string);
				}
				result.put(chave, value);
			}

		} catch (SQLException sqe) {
			sqe.printStackTrace();
		} catch (ClassNotFoundException e) {
```

```
                    e.printStackTrace();
                }

                finally {
                    try {
                        DBConnector.cleanup(conect, prepStmt);
                    } catch (SQLException sqe) {
                        sqe.printStackTrace();
                    }
                }
                return result;
            }

            public String getValue(String chave) {
                StringBuilder query = new StringBuilder();
                String consulta;
                PreparedStatement prepStmt = null;
                Connection conect = null;
                ResultSet resultSet;
                String value = null;

                try {
                    conect = DBConnector.getConnection();

                    query.append("SELECT VALORES FROM " + this.tableName +
"_blocks_standart_8K ");
                    query.append("WHERE CHAVE = ?");

                    consulta = query.toString();

                    prepStmt = conect.prepareStatement(consulta);
                    prepStmt.setString(1, chave);
                    resultSet = prepStmt.executeQuery();

                    while (resultSet.next()) {
                        value = resultSet.getString(1);
                    }
```

```
                } catch (SQLException sqe) {
                        sqe.printStackTrace();
                } catch (ClassNotFoundException e) {
                        e.printStackTrace();
                }

                finally {
                        try {
                                DBConnector.cleanup(conect, prepStmt);
                        } catch (SQLException sqe) {
                                sqe.printStackTrace();
                        }
                }
                return value;
        }


        public void setTableName(String tableName) {
                this.tableName = tableName;
        }
}
```

```
package br.com.pSimUtils.bd.dao;

public class TrigramBlockingDAO implements INGramDAO{

	@Override
	public void insereEntradaTabela(String chave, String valor) {
		// TODO Auto-generated method stub

	}

	@Override
	public void atualizaEntradaTabela(String chave, String valor) {
		// TODO Auto-generated method stub

	}

	@Override
	public String getValue(String chave) {
		// TODO Auto-generated method stub
		return null;
	}

	@Override
	public void setTableName(String tableName) {
		// TODO Auto-generated method stub
```

```
	}
}

	package br.com.pSimUtils.db.manager;

	import java.sql.Connection;
	import java.sql.DriverManager;
	import java.sql.SQLException;
	import java.sql.Statement;

	import org.postgresql.Driver;

	import snaq.db.ConnectionPool;



	public class DBConnector {
		static String urlConnection = "jdbc:postgresql://127.0.0.1:5432/pbsim";
		static String usuario = "postgres";
		static String senha = "1234";
		private static ConnectionPool cpInstance;

		public static Connection getConnection() throws SQLException,
ClassNotFoundException {
			Connection conn = null;
			try {
				conn = getInstance().getConnection(15);
			} catch (InstantiationException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			} catch (IllegalAccessException e) {
				// TODO Auto-generated catch block
				e.printStackTrace();
			}
			return conn;
		}
```

```
		public static void closeConnection (Connection conection) throws SQLException {
			if(conection != null)
			conection.close();

		}

		public static void closeStatement (Statement stmt) throws SQLException {
			if(stmt != null)
			stmt.close();
		}

		public static void cleanup (Connection conn, Statement stmt) throws SQLException {
			closeStatement(stmt);
			closeConnection(conn);
		}

		public static ConnectionPool getInstance() throws ClassNotFoundException,
SQLException, InstantiationException, IllegalAccessException {
			if(cpInstance == null) {

				@SuppressWarnings("rawtypes")
				Class klass = Class.forName("org.postgresql.Driver");
				Driver driver = (Driver)klass.newInstance();
				DriverManager.registerDriver(driver);
				cpInstance = new ConnectionPool("mainPool", 200, 200, 20,
urlConnection, usuario, senha);
				return cpInstance;
			}
			return cpInstance;
		}
	}

	package br.com.psim.executor

	import java.util.Date
	import java.util.List
	import scala.collection.JavaConversions
```

```
import scala.collection.immutable.HashMap
import scala.collection.mutable.ArrayBuffer
import scala.collection.mutable.Buffer
import scala.collection.parallel.ForkJoinTaskSupport
import scala.collection.parallel.TaskSupport
import scala.collection.parallel.TaskSupport
import scala.collection.parallel.mutable.ParArray
import br.com.pSim.blocking.ngram.NgramBlocking
import br.com.pSimUtils.bd.dao.BigramBlockingDAO
import br.com.pSimUtils.bd.dao.DataCollectorDAO
import br.com.pSimUtils.bd.dao.StandartBlockingDAO
import br.com.psim.traits.LogHelper
import uk.ac.shef.wit.simmetrics.similaritymetrics.BlockDistance
import uk.ac.shef.wit.simmetrics.similaritymetrics.DiceSimilarity
import uk.ac.shef.wit.simmetrics.similaritymetrics.EuclideanDistance
import uk.ac.shef.wit.simmetrics.similaritymetrics.Jaro
import uk.ac.shef.wit.simmetrics.similaritymetrics.Levenshtein
import scala.collection.parallel.mutable.ParMap

object ParallelExecutor extends LogHelper {

 //Funcoes de similaridade que podem ser usadas...
 val levenshtein: Levenshtein = new Levenshtein();
 val bd: BlockDistance = new BlockDistance;
 val ed: EuclideanDistance = new EuclideanDistance;
 val jr: Jaro = new Jaro;
 val ds: DiceSimilarity = new DiceSimilarity;

 var bgCache: ParMap[String, java.util.List[String]] = ParMap[String, java.util.List[String]]();
 var stCache: ParMap[String, java.util.List[String]] = ParMap[String, java.util.List[String]]();

 def parallelSimilarityFuncionExecutor(data: Iterable[String], minScore: Double) {

  val dataPar = data.par;
  dataPar.tasksupport = new ForkJoinTaskSupport(new
scala.concurrent.forkjoin.ForkJoinPool(4));
  println(dataPar.size);
  val dataInicio: Date = new Date();
```

```
            logger.info("Data inicio: " + dataInicio);
            println(dataInicio.toString);
            for (value <- dataPar) yield {
              for (nextValue <- dataPar) yield {
                compareValues(value, nextValue, minScore);
              }
            }
            val dataFim: Date = new Date();
            logger.info("Processo finalizado!!" + dataFim);
            println("Processo finalizado!!");
            println(dataFim);
            val tempoTotal = (dataFim.getTime() / 1000) - (dataInicio.getTime() / 1000);
            println("Tempo total = " + tempoTotal + " segundos");
          }

          def parallelSimilarityBigramBlockingFuncionExecutor(data: Buffer[String], minScore: Double) {
            val dataPar = data.par;
            dataPar.tasksupport = new ForkJoinTaskSupport(new
scala.concurrent.forkjoin.ForkJoinPool(4));
            println(dataPar.size);
            val dataInicio: Date = new Date();
            //    var i = 0;
            println(dataInicio);
            logger.info("Data inicio: " + dataInicio);
            for (frase <- dataPar) yield {

              val palavrasSplit = getPalavrasSplited(frase).par;
              palavrasSplit.tasksupport = new ForkJoinTaskSupport(new
scala.concurrent.forkjoin.ForkJoinPool(4));
              for (palavra <- palavrasSplit) yield {
                //                logger.info("palavras separadas " + palavrasMesmaChave.size);
                val palavrasMesmaChave = getPalavrasMesmaChaveBG(palavra).par;
                palavrasMesmaChave.tasksupport = new ForkJoinTaskSupport(new
scala.concurrent.forkjoin.ForkJoinPool(4));
                for (value <- palavrasMesmaChave) yield {
                  //                  logger.info("iterou sobre as palavras");
                  compareValues(value, frase, minScore);
                }
```

```
    }
   }
   val dataFim: Date = new Date();
   logger.info("Processo finalizado!!" + dataFim);
   //    println("Iterou " + i + " vezes");
   println("Processo finalizado!!");
   println(dataFim);
   val tempoTotal = (dataFim.getTime() / 1000) - (dataInicio.getTime() / 1000);
   println("Tempo total = " + tempoTotal + " segundos");
  }

  def compareValues(value: String, frase: String, minScore: Double) {
   if (value != null && frase != null) {
    val score = levenshtein.getSimilarity(value, frase);
    val score1 = bd.getSimilarity(value, frase);
    val score2 = ed.getSimilarity(value, frase);
    val score3 = jr.getSimilarity(value, frase);
    if (score >= minScore && score1 >= minScore && score2 >= minScore && score3 >=
minScore) {
     //                                    logger.info("score = " + score);
     logger.info("Valores = " + value + " " + frase + " Levenshtein Score = " + score);
     logger.info("Valores = " + value + " " + frase + " BlockDistance Score = " + score1);
     logger.info("Valores = " + value + " " + frase + " EuclideanDistance Score = " + score2);
     logger.info("Valores = " + value + " " + frase + " Jaro Score = " + score3);
    }
   }
  }

  def getPalavrasMesmaChaveBG(palavra: String): Buffer[String] =
   {

    val ngramBlock = new NgramBlocking(null, 0.9, 2, "ruas");
    var palavrasMesmaChave: Buffer[String] = Buffer[String]();
    val chaves = JavaConversions.asScalaBuffer(ngramBlock.getKeys(palavra));
    for (chave <- chaves) {
     palavrasMesmaChave.appendAll(getBGCacheValue(chave));
    }
    palavrasMesmaChave;
```

```
  }

  def getBGCacheValue(chave: String): Buffer[String] =
  {
    var palavrasChave: Buffer[String] = Buffer[String]();
    if (bgCache.contains(chave)) {
      palavrasChave = JavaConversions.asScalaBuffer(bgCache.get(chave).get);
    } else {
      println("Não localizou a chave " + chave + " no cache!!");
    }
    palavrasChave;
  }

  def getPalavrasSplited(frase: String): Array[String] = {
    var palavrasSplit: Array[String] = frase.split(" ");
    palavrasSplit;
  }

  def parallelSimilarityStandartBlockingFuncionExecutor(data: Buffer[String], minScore: Double)
{
    val dataPar = data.par;
    var palavrasMesmaChave: Buffer[String] = Buffer[String]()
    println(dataPar.size);
    val dataInicio: Date = new Date();
    println(dataInicio);
    logger.info("Data inicio: " + dataInicio);
    for (frase <- dataPar) yield {
      palavrasMesmaChave                                                                    =
getPalavrasMesmaChaveST(String.valueOf(frase.toUpperCase.charAt(0)));
      for (palavra <- palavrasMesmaChave.par) yield {
        compareValues(frase, palavra, minScore);
      }
    }
    val dataFim: Date = new Date();
    logger.info("Processo finalizado!!" + dataFim);
    println("Processo finalizado!!");
    println(dataFim);
    val tempoTotal = (dataFim.getTime() / 1000) - (dataInicio.getTime() / 1000);
```

```
println("Tempo total = " + tempoTotal + " segundos");
}

def getPalavrasMesmaChaveST(chave: String): Buffer[String] =
 {

  var palavrasChave: Buffer[String] = Buffer[String]();
  if (stCache.contains(chave)) {
   palavrasChave = JavaConversions.asScalaBuffer(stCache.get(chave).get);
  } else {
   println("Não localizou a chave " + chave + " no cache!!");
  }
  palavrasChave;
 }

def loadCacheBigram() {
 println("Carregando o cache");
 val bgDAO = new BigramBlockingDAO();
 bgDAO.setTableName("instituicoes");
 bgCache = JavaConversions.mapAsScalaMap(bgDAO.loadCache(8)).par;
 println("Cache blocagem BIGRAM carregado com " + bgCache.size + " chaves!!!!");
}

def loadCacheStandart() {
 println("Carregando o cache");
 val stDAO = new StandartBlockingDAO();
 stDAO.setTableName("instituicoes");
 stCache = JavaConversions.mapAsScalaMap(stDAO.loadCache(8)).par;
 println("Cache blocagem STANDART carregado com " + stCache.size + " chaves!!!!");
}

def main(args: Array[String]) {
 //   val cidades = DataCollectorDAO.getCidades();
 val instituicoes = DataCollectorDAO.getInstituicoes();
 //   val ruas = DataCollectorDAO.getRuas();
 loadCacheBigram();
 loadCacheStandart();
//      parallelSimilarityFuncionExecutor(JavaConversions.asScalaBuffer(instituicoes), 0.8);
```

```
            //
parallelSimilarityBigramBlockingFuncionExecutor(JavaConversions.asScalaBuffer(instituicoes), 0.8);

parallelSimilarityStandartBlockingFuncionExecutor(JavaConversions.asScalaBuffer(instituicoes), 0.8);
             }

            }

            package br.com.psim.executor

            import java.util.Date
            import java.util.HashMap
            import java.util.List
            import java.util.Map

            import scala.Array.apply
            import scala.Array.canBuildFrom
            import scala.collection.JavaConversions
            import scala.collection.mutable.ArrayBuffer
            import scala.collection.mutable.Buffer
            import scala.collection.mutable.Buffer.apply

            import br.com.pSim.blocking.ngram.NgramBlocking
            import br.com.pSimUtils.bd.dao.BigramBlockingDAO
            import br.com.pSimUtils.bd.dao.DataCollectorDAO
            import br.com.pSimUtils.bd.dao.StandartBlockingDAO
            import br.com.psim.traits.LogHelper
            import uk.ac.shef.wit.simmetrics.similaritymetrics.BlockDistance
            import uk.ac.shef.wit.simmetrics.similaritymetrics.DiceSimilarity
            import uk.ac.shef.wit.simmetrics.similaritymetrics.EuclideanDistance
            import uk.ac.shef.wit.simmetrics.similaritymetrics.Jaro
            import uk.ac.shef.wit.simmetrics.similaritymetrics.Levenshtein

            object SingleExecutor extends LogHelper {

             //Funcoes de similaridade que podem ser usadas...
             val levenshtein: Levenshtein = new Levenshtein();
             var bd: BlockDistance = new BlockDistance;
```

```
var ed: EuclideanDistance = new EuclideanDistance;
var jr: Jaro = new Jaro;
var ds: DiceSimilarity = new DiceSimilarity;
var bgCache: Map[String, List[String]] = new HashMap[String, List[String]]();
var stCache: Map[String, List[String]] = new HashMap[String, List[String]]();

def seqSimilarityFuncionExecutor(data: Iterable[String], minScore: Double) {

 val dataPar = data;
 println(dataPar.size);
 val dataInicio: Date = new Date();
 logger.info("Data inicio: " + dataInicio);
 println(dataInicio.toString);
 for (value <- dataPar) {
  for (nextValue <- dataPar) {
   compareValues(value, nextValue, minScore);
  }
 }
 val dataFim: Date = new Date();
 logger.info("Processo finalizado!!" + dataFim);
 println("Processo finalizado!!");
 println(dataFim);
 val tempoTotal = (dataFim.getTime()/1000) - (dataInicio.getTime() / 1000);
 println("Tempo total = " + tempoTotal + " segundos");
}


def singleBigramComparation(data: Buffer[String], minScore: Double) {
 val dataPar = data;
 println(dataPar.size);
 val dataInicio: Date = new Date();
 //    var i = 0;
 println(dataInicio);
 logger.info("Data inicio: " + dataInicio);
 for (frase <- dataPar) yield {
  val palavrasSplit = getPalavrasSplited(frase);
  for (palavra <- palavrasSplit) yield {
   //                    logger.info("palavras separadas " + palavrasMesmaChave.size);
```

```
            val palavrasMesmaChave = getPalavrasMesmaChave(palavra)
            for (value <- palavrasMesmaChave) yield {
              //                    logger.info("iterou sobre as palavras");
              compareValues(value, frase, minScore);
            }
          }
        }
        val dataFim: Date = new Date();
        logger.info("Processo finalizado!!" + dataFim);
        //    println("Iterou " + i + " vezes");
        println("Processo finalizado!!");
        println(dataFim);
        val tempoTotal = (dataFim.getTime()/1000) - (dataInicio.getTime() / 1000);
        println("Tempo total = " + tempoTotal + " segundos");
      }

      def compareValues(value: String, frase: String, minScore: Double) {
        if (value != null && frase != null) {
          val score = levenshtein.getSimilarity(value, frase);
          val score1 = bd.getSimilarity(value, frase);
          val score2 = ed.getSimilarity(value, frase);
          val score3 = jr.getSimilarity(value, frase);
          if (score >= minScore && score1 >= minScore && score2 >= minScore && score3 >=
minScore) {
            logger.info("Valores = " + value + " " + frase + " Levenshtein Score = " + score);
            logger.info("Valores = " + value + " " + frase + " BlockDistance Score = " + score1);
            logger.info("Valores = " + value + " " + frase + " EuclideanDistance Score = " + score2);
            logger.info("Valores = " + value + " " + frase + " Jaro Score = " + score3);
          }
        }
      }

      def getPalavrasMesmaChave(palavra: String): Buffer[String] =
        {

          val ngramBlock = new NgramBlocking(null, 0.9, 2, "ruas");
          var palavrasMesmaChave: Buffer[String] = Buffer[String]();
          val chaves = JavaConversions.asScalaBuffer(ngramBlock.getKeys(palavra));
```

```
    for (chave <- chaves) {
      palavrasMesmaChave.appendAll(getCacheValue(chave));
    }
    palavrasMesmaChave;
  }

  def getCacheValue(chave: String): Buffer[String] =
  {
    var palavrasChave: Buffer[String] = Buffer[String]();
    if (bgCache.containsKey(chave)) {
      palavrasChave = JavaConversions.asScalaBuffer(bgCache.get(chave));
    } else {
      println("Não localizou a chave " + chave + " no cache!!");
    }
    palavrasChave;
  }

  def getPalavrasSplited(frase: String): Array[String] = {
    var palavrasSplit: Array[String] = frase.split(" ");
    palavrasSplit;
  }

  def singleSimilarityStandartBlockingFuncionExecutor(data: Buffer[String], minScore: Double) {
    val dataPar = data;
    var palavrasMesmaChave: Buffer[String] = Buffer[String]()
    println(dataPar.size);
    val dataInicio: Date = new Date();
    println(dataInicio);
    logger.info("Data inicio: " + dataInicio);
    for (frase <- dataPar) yield {
      palavrasMesmaChave                                                    =
getPalavrasMesmaChaveST(String.valueOf(frase.toUpperCase.charAt(0)));
      for (palavra <- palavrasMesmaChave) yield {
        compareValues(frase, palavra, minScore);
      }
    }
    val dataFim: Date = new Date();
    logger.info("Processo finalizado!!" + dataFim);
```

```
  println("Processo finalizado!!");
  println(dataFim);
  val tempoTotal = (dataFim.getTime()/1000) - (dataInicio.getTime() / 1000);
  println("Tempo total = " + tempoTotal + " segundos");
 }

 def getPalavrasMesmaChaveST(chave: String): Buffer[String] =
  {

    var palavrasChave: Buffer[String] = Buffer[String]();
    if (stCache.containsKey(chave)) {
      palavrasChave = JavaConversions.asScalaBuffer(stCache.get(chave));
    } else {
      println("Não localizou a chave " + chave + " no cache!!");
    }
    palavrasChave;
  }

 def loadCache() {
  println("Carregando o cache");
  val bgDAO = new BigramBlockingDAO();
  bgDAO.setTableName("instituicoes");
  bgCache = bgDAO.loadCache(8);
  println("Cache blocagem BIGRAM carregado com " + bgCache.size + " chaves!!!!");
 }

 def loadCacheStandart() {
  println("Carregando o cache");
  val stDAO = new StandartBlockingDAO();
  stDAO.setTableName("instituicoes");
  stCache = stDAO.loadCache(8);
  println("Cache blocagem STANDART carregado com " + stCache.size + " chaves!!!!");
 }

 def main(args: Array[String]) {
//    val ruas = DataCollectorDAO.getRuas();
   val instituicoes = DataCollectorDAO.getInstituicoes();
   loadCache();
```

```
            loadCacheStandart();
          //   seqSimilarityFuncionExecutor(JavaConversions.asScalaBuffer(instituicoes), 0.8);
          //   singleBigramComparation(JavaConversions.asScalaBuffer(instituicoes), 0.8);
            singleSimilarityStandartBlockingFuncionExecutor(JavaConversions.asScalaBuffer(instituicoes),
0.8);
         }


        }
```

```
package br.com.psim.traits

import org.apache.log4j.Logger

trait LogHelper {

       val loggerName = this.getClass.getName
     lazy val logger = Logger.getLogger(loggerName)
       }
```

8. Apêndice 2 – Artigo

Execução de Experimentos com Funções de Similaridade em um Ambiente Paralelo

Gustavo de Geus' Carina F. Dorneles

Departamento de Informática e Estatística – Universidade Federal de Santa Catarina (UFSC)
Caixa Postal 476 – 88.040-900 – Florianópolis – SC – Brazil
gustavodegeus@gmail.com | dorneles@inf.ufsc.br

Resumo. *Com o volume e falta de padronização de dados crescendo a cada dia, o uso de funções de similaridade torna-se cada vez mais importante para quem deseja gerenciar esse grande volume de dados, porém a execução dessas funções para uma grande massa de dados se torna um processo altamente custoso computacionalmente. Uma das propostas para tornar possível a execução dessas funções de similaridade em grandes volumes de dados é a utilização de programação paralela em conjunto com técnicas de blocagem. A proposta apresentada nesse artigo é a definição de uma arquitetura que utiliza os alguns conceitos de programação paralela e de blocagem de dados para que a realização experimentos, realizados com o intuito de mostrar os benefícios que podem ser obtidos através da programação paralela juntamente com a blocagem de dados.*

1. Introdução

Um dos maiores desafios no gerenciamento de dados é tratar o grande volume de dados que vem sendo gerado a cada dia. Para piorar o cenário, esses dados são gerados de forma variada por diferentes usuários e softwares, fazendo com que surjam inúmeras replicações de dados, ou seja, um mesmo objeto do mundo real representado de maneiras diferentes no mundo digital. Para descobrir que dois valores de dados tratam do mesmo objeto do mundo real, podemos fazer o uso de funções de similaridade, porém a utilização das funções de similaridade aplicadas em um grande volume de dados na tentativa de solucionar o problema é algo bastante oneroso computacionalmente.

Uma das alternativas para contornar esse gargalo de desempenho é utilizar processamento paralelo para a execução das funções de similaridade. Trabalhando sobre o paradigma de programação paralela, é possível programar a execução de um bloco de instruções de forma fragmentada, onde um processo pai é divido em tarefas menores e, cada uma dessas pequenas tarefas é processada de forma independente. Esse modelo de programação faz com que se tenha um ganho de performance considerável, além de utilizar melhor os recursos computacionais disponíveis no ambiente onde a aplicação está sendo executada.

A maior complexidade em utilizar a abordagem de programação paralela está na dificuldade em se controlar esses pequenos processos. O resultado de cada execução deve ser concatenado da maneira lógica para que ao final o resultado gerado seja o esperado além de ser necessário o controle de conclusão de tarefa. No momento em que um dos processos filhos é concluído, o recurso no processador deve ser liberado para que outro processo filho seja executado. Em resumo, manter a sincronia dos pequenos processos assim como a gestão dos recursos são, ainda, os dois pontos que tornam a programação paralela mais complexa.

O cenário composto pelo grande volume de registros de dado não padronizados, agregados a necessidade de tratamentos utilizando funções de similaridade de maneira mais eficiente, incentivaram estudos em dois sentidos. Uma das propostas é construir blocos de dados e a outra é fazer a execução do processo de forma paralela. Foram feitos alguns estudos juntando os dois conceitos *P-Swoosh* [KAWAI et. al. 2006] e *P-Canopy* [DAL BIANCO et. al. 2009], conseguindo-se blocar dados de maneira mais otimizada.

A idéia principal desse trabalho é explorar a possibilidade de execução das funções de similaridades utilizando programação paralela (*multi-thread)* de maneira simples, e eficiente. Para isso foram realizados diversos experimentos, com diferentes funções de similaridade, tipos de dados diferentes e utilizando ou não técnicas de blocagem a fim de se obter resultados que possam realmente demonstrar qual a real diferença entre a adoção do processamento paralelo das funções de similaridade.

Para uma melhor compreensão esse artigo foi dividido da seguinte forma: na seção 2 são apresentados alguns conceitos básicos que foram utilizados no trabalho. A seção 3 descreve brevemente alguns trabalhos que tiveram relação com o trabalho proposto. Na seção 4 são mostradas as abordagens referentes à blocagem de dados, funções de similaridade e programação paralela utilizadas no P-BSim (nome dados ao projeto). A seção 5 visa falar brevemente de como foram implementados os conceitos citados na seção 4 em questões tecnológicas. As seções 6 e 7 tratam dos experimentos realizados e os resultados obtidos respectivamente. Por fim, a seção 8 apresenta as conclusões que puderam ser obtidas ao final do trabalho e, a seção 9 cita as principais referencias bibliográficas utilizadas.

2. Conceitos básicos

Antes de apresentar a proposta, alguns conceitos gerais sobre funções de similaridade, blocagem de dados e paralelização de algoritmos serão apresentados. Funções de similaridade são aplicadas em situações onde o uso do operador de igualdade não é adequado, pois os valores comparados apresentam diferentes representações, mesmo quando se referem ao mesmo objeto do mundo real, por exemplo, quando ocorre duplicação de dados como nomes próprios, datas, endereços, nomes de cidades e países, entre outros. Esta questão tem motivado pesquisa nas mais diversas áreas e contextos, desde questões relacionadas à consulta sobre bases de dados com mais de uma representação do mesmo objeto até a integração de fontes que possuem dados referentes ao mesmo domínio.

As técnicas de blocagem têm como objetivo principal limitar o número de comparações realizadas no processo de deduplicação, tendo em vista que no pior dos casos, o quadrático ou produto cartesiano (PC), são realizadas comparações com dados

com pouca ou nenhuma relação. Fazendo uma analogia com sistemas de banco de dados, a blocagem funciona como um processo de indexação por similaridade. Nesse artigo foram utilizadas as técnicas de blocagem *Standart Index e Bigram Index*. A técnica *Standart Index* utilizou como predicado de blocagem apenas a primeira letra de cada palavra que compunha o registro, enquanto a *Bigram Index* teve a abordagem padrão definida na literatura, tendo como predicado de blocagem todas as combinações entre os bigramas das palavras do registro. A implementação da ambos os modelos de blocagem seguiu o modelo de algoritmo detalhado na literatura, [Baxter et al. 2003], [Bilenko et al. 2006] e [Christen 2007] respectivamente.

A programação paralela vem sendo aprimorada em diversas linguagens de programação, muitas delas já abstraíram o controle de *threads*, deixando o desenvolvedor mais focado nas regras de negócio do problema do que no controle de *threads*. Uma das linguagens que tem um suporte à programação paralela bastante desenvolvida é o Scala, uma linguagem que trabalha sob o paradigma funcional ou orientado a objetos. Em virtude de o Scala possuir uma integração transparente com o JAVA, muitas empresas estão adotando o Scala quando se necessita de desempenho e escalabilidade. Um exemplo é o *Twitter*, que mudou toda a sua fila de mensagens que antes era feita em *Ruby* para Scala [Scala in the Enterprise, 2012].

3. Trabalhos relacionados

[Kawai et. al. 2006] – Tendo como nome “P-Swoosh”, esse trabalho foca no problema de Resolução de Entidades (ER), tendo como objetivo trabalhar em cima de duas funções: *match* e *merge*, ou combinar e fundir. A função *match* é encarregada de identificar dados duplicados, enquanto a *merge* mescla dados semelhantes a fim de inseri-los em um mesmo contexto.

[Santos, et. al. 2007] – Tendo como nome “A Scalable Parallel Deduplication Algorithm”, é um trabalho que tem como foco uma estratégia para a resolução baseada em quatro filtros, *ReaderComparator, Blocking, Classifier, e o Merger*. O *ReaderComparator* é encarregado tanto de definir a chave de blocagem quanto de comparar os dados, o *Blocking* é encarregado de filtrar cada chave de blocagem criada e gerenciar a necessidade ou não de criar um novo bloco de dados. Em auxílio a esses dados vem o filtro de *Merger* que tem como objetivo evitar a geração de pares redundantes mantendo uma *hashtable* com a combinação de todos os identificadores gerados pelo *ReaderComparator*. Por fim, o *Classifier* é encarregado de verificar os resultados gerados pela comparação feita e classificar os pares de dados como relacionados e não relacionados.

[Gonçalvez, et. al. 2008] – Tendo como nome “Avaliação de técnicas paralelas de blocagem para resolução de entidades e deduplicação”, esse trabalho tem como foco principal a utilização de técnicas de blocagem, que agrupam os registros com alguma semelhança com base em métricas básicas. Sendo que a execução do procedimento é realizada em um ambiente paralelo, a fim de se verificar quais as reais melhoras quando se trabalha em uma arquitetura paralela distribuída.

A diferença entra o presente trabalho (P-BSim) e o *P-Swoosh* [Kawai et. al. 2006] é que o P-BSim visa demonstrar o funcionamento das funções de similaridade em um ambiente paralelo de maneira simples, utilizando ou não dados blocados. O *P-*

Swoosh visa uma abordagem genérica para a resolução dos problemas de deduplicação, enquanto o P-BSim tem como objetivo mensurar o ganho real da paralelização de execuções de similaridade e blocagem de dados.

Quanto ao [Santos, et. al. 2007], o P-BSim se difere na etapa paralelizada e também por abordar além das técnicas de blocagem a execução de funções de similaridade para a resolução do problema da deduplicação. Em [Santos, et. al. 2007] são propostos alguns modelos para paralelizar a blocagem, porém o P-BSim visou paralelizar apenas a execução das funções de similaridade, partindo de dados que podem ou não estarem blocados.

O trabalho descrito em [Gonçalvez, et. al. 2008] é o que mais se assemelha ao trabalho proposto, com o diferencial que em [Gonçalvez, et. al. 2008] é sugerido um novo algoritmo para o problema de deduplicação enquanto no P-BSim é mensurado o quanto o processo de blocagem juntamente com a paralelização das funções de similaridade beneficia o processo de deduplicação como um todo.

4. P-BSim

Essa seção tem como objetivo apresentar o funcionamento do P-BSim (*Parallel Blocking Similarity*), um ambiente que permite a avaliação de desempenho de funções de similaridade executadas de maneira multiprocessada sob um conjunto de dados blocados. A principal contribuição obtida com o uso desse ambiente foi a de se ter medições significativas quanto à vantagem de realizar o processo de deduplicação de forma paralela, utilizando assim o máximo dos recursos computacionais disponíveis.

4.1 Visão geral

A execução do P-BSim pode ser dividida em duas etapas principais, onde uma delas é a etapa de blocagem de dados e a outra o casamento de dados usando funções de similaridade efetivamente. Na parte de blocagem de dados existe uma interação maior entre bases de dados, contendo os registros de texto de variados tipos. Na execução das funções, os dados blocados são lidos e o resultado da execução das funções para cada registro é armazenada em um arquivo de texto, usado posteriormente na fase de experimentos.

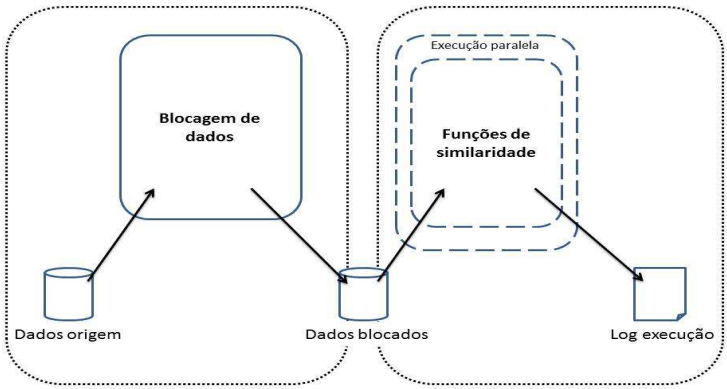


Figura 1 - Esquema geral do P-BSim

4.2 – Blocagem de dados

No P-BSim, o processo de blocagem de forma geral é definido conforme a Figura 2. Quanto às estratégias utilizadas, o *Standart Blocking* (ST) e *Bigram Blocking* (BI), apresentam diferenças em sua estrutura apenas na etapa demarcada em verde, que consiste na obtenção das chaves de blocagem para o registro.

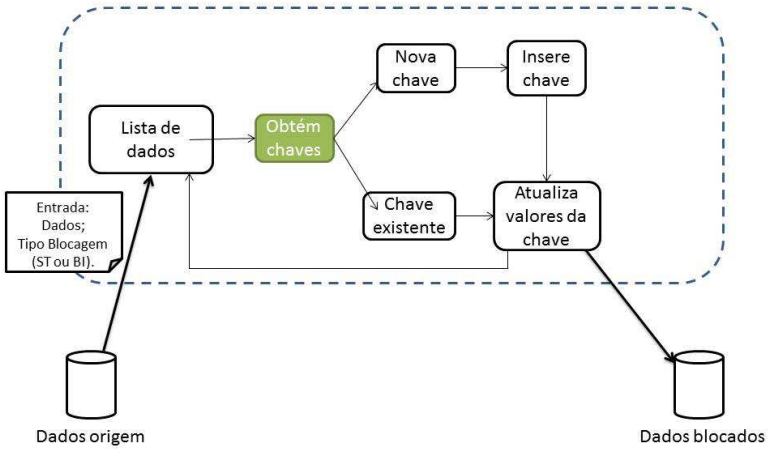


Figura 2 - Esquema geral da blocagem de dados do P-BSim

A partir de uma dada base de dados, são lidos os registros que devem ser blocados. Com esses registros em memória, são iniciadas iterações sobre a lista de dados e para cada valor são realizados os seguintes processos:

- **obtenção de chave:** esse processo destacado na Figura 2 é a única parte que sofre alteração, dependendo da estratégia de blocagem utilizada no processo. Para esse trabalho o foco foi voltado somente nas estratégias *Bigram Blocking* e *Standart Blocking.*
- **verificação de chave:** com a chave em mãos é realizada uma verificação para garantir se é uma chave já existente ou não.
- **inserção de uma nova chave:** caso a chave em questão não exista no repositório de dados blocados, ela é inserida no banco de dados tendo como único valor o próprio registro que a gerou.
- **atualização de valores para uma chave:** por fim, caso a chave gerada já exista no repositório, é adicionada uma nova entrada para essa chave com os dados em questão.

Dessa forma, ao final desse processo tem-se uma tabela com duas colunas, onde o índice principal será a chave, e os valores serão todas as palavras que compartilham dessa chave.

4.3 - Processamento paralelo de funções de similaridade

O P-BSim implementa a abordagem de paralelização no processamento das funções de similaridade, fazendo com que seja possível distribuir a execução da função em diferentes processos, alimentando o mesmo arquivo de *log* com os resultados das comparações. Isso torna viável a exploração dos múltiplos núcleos que os novos processadores vêm oferecendo. O funcionamento básico da execução das funções é representado na Figura 3 para dados blocados, e na Figura 4 para dados não blocados.

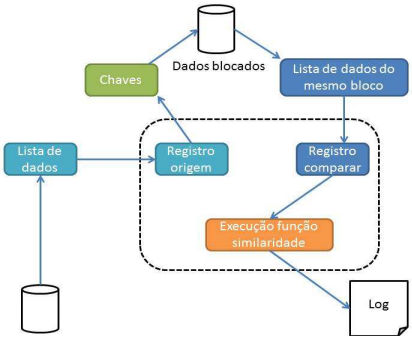


Figura 4 - Execução de funções em dados blocados

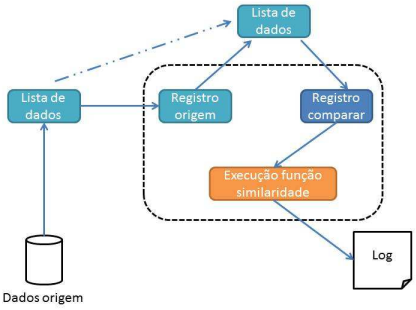


Figura 3 - Execução de funções em dados não blocados (Produto cartesiano)

No P-BSim, as funções de similaridade podem ser executadas sobre dados blocados ou não. Para a execução de testes utilizando os dados blocados, o processo de blocagem em cima da base de dados original deve ser feito de forma independente. Dessa forma, antes da execução dos testes, o repositório com os dados blocados já está completo e pode ser consumido na fase de teste. Essa abordagem foi necessária para que fosse possível avaliar o quanto a blocagem dos dados influenciaria no resultado final tanto em questão de desempenho quanto de qualidade.

Grande parte da complexidade do modelo de programação é abstraída pela API utilizada, as Parallel Collections do SCALA. Essa API contém estruturas de dados que operam e controlam todas as operações de acesso e iterações de forma paralela. Isso fez com que o grau de complexidade na implementação do P-BSim caísse bastante, permitindo que durante todo o processo todos os núcleos do processador estivessem sendo utilizados.

5 – Implementação

Na parte de blocagem, a implementação foi toda feita utilizando a linguagem de programação JAVA juntamente com banco de dados Postgres SQL. Já na parte de execução das funções de similaridade, foi utilizada a linguagem de programação SCALA, armazenando os resultados em um arquivo de *LOG* gerenciado pela própria aplicação.

Para a blocagem de dados, o algoritmo que realizou a extração das chaves e as operações com o banco de dados utilizando o driver JDBC para acessar as tabelas no

banco de dados Postgres SQL. Toda a execução desse processo foi feita em ambiente sequencial, sem utilizar qualquer estrutura de dados com suporte a operações paralelas ou algum recurso que tornasse a execução da blocagem paralela.

Para a execução das funções de similaridade, foi feita a implementação do algoritmo em SCALA. Dessa forma, foi possível implementar a execução das funções de similaridade utilizando as estruturas de dados paralelas do SCALA (Parallel Collections) junto com estruturas de dados simples do próprio SCALA ou do JAVA. Além das estruturas de dados, foram utilizadas duas bibliotecas. A biblioteca Log4J forneceu um mecanismo para escrever o resultado das funções de similaridade de forma assíncrona, não afetando a performance dos testes. Já a biblioteca SIMMETRICS, ofereceu as implementações de todas as funções de similaridade utilizadas.

6 – Experimentos

Os testes foram executados em um computador possuindo o seguinte *hardware*: processador *Intel Core i5 2410M* (quatro núcleos) com *4GB* de memória *RAM*. Repartindo os dados em subconjuntos, foi possível montar repositórios com um volume de dados variando desde um valor mínimo, até todos os registros do repositório de dados original. Quanto ao parâmetro de entrada, como apresentado na seção anterior, o único parâmetro que o algoritmo leva em consideração é o limiar que a função deve retornar para que a comparação seja catalogada no log. O valor foi fixado em 0.9 para todos os testes. Por fim, quanto às funções de similaridade utilizadas, foram selecionadas as seguintes funções dentro das disponíveis na biblioteca *Simetrics: Levenshtein, BlockDistance, EuclideanDistance* e *Jaro.* Todas as funções foram executadas ao mesmo tempo, dentro do mesmo *"loop"* de execução, para que fosse possível tirar maior proveito do paralelismo.

Foram utilizados diferentes domínios de dados, que vão desde nomes de cidades até nomes de instituições de ensino. Cada uma das bases avaliadas sofreu uma análise antes da execução dos testes para que, com o conhecimento do domínio, fosse possível chegar a conclusões sobre quanto o tipo de dados avaliado interfere na execução das funções de similaridade. Isso viabilizou a obtenção de resultados em cenários bastante heterogêneos, tendo em vista que para domínios diferentes, a execução do algoritmo pode ser mais trabalhosa ou não.

7 – Resultados

Tendo os artefatos devidamente desenvolvidos, deu-se início aos testes preliminares, que foram separados em quatro grandes grupos. Sendo eles:

- dados não blocados sem a utilização de programação paralela
- dados não blocados utilizando programação paralela
- dados blocados utilizando programação paralela
- dados blocados sem a utilização de programação paralela.

Os testes também foram separados em grupos tendo como critério para separação o tipo de dado avaliado. Dessa forma, foi possível avaliar os resultados de forma individual e, realizar um comparativo entre os diferentes tipos de testes a fim de

verificar qual o impacto provocado pelos diferentes tipos de dados. Os testes foram classificados da seguinte forma:

- **Teste 1 (Teste cid.)** – testes realizados sob uma base contendo nome de cidades brasileiras. Antes da execução foi realizada uma pequena auditoria na base através de consultas *SQL* e visto que existiam diversos dados repetidos e erros léxicos. No total, a base possuía 10830 registros, os quais foram divididos conforme o fator de amostragem de dados explicado à cima.
- **Teste 2 (Teste rua)** – testes realizados sob uma base de dados contendo nome de ruas. Na verificação realizada nos dados foi visto que os nomes de ruas normalmente são "registros" maiores, contendo três ou mais palavras. Uma inconsistência recorrente nessa base era a denominação da via, onde em alguns registros estava apenas "R", outro "R.", outros "Rua" e outras descrições para identificar que a via era uma rua comum. O mesmo ocorria para avenidas, travessas, servidões e outras denominações de vias. No total a base possuía 9895 registros.
- **Teste 3 (Teste inst.)** – teste realizado tendo como entrada de dados nomes de instituições de ensino brasileiras, sendo elas Universidades, Escolas técnicas, colégios e alguns outros tipos de instituições. Na verificação de domínio do dado foi visto que os registros dessa tabela eram os maiores se comparados com os utilizados nos outros testes (1 e 2) porque normalmente antes do nome, era indicado o tipo da instituição de ensino. Uma das inconsistências que foram notados foi justamente a denominação do tipo de instituição, que era escrita de diversas formas como, por exemplo, "ESC", "ESC.", "ESCOLA", entre outras para identificar que era uma escola. Essa base possuía um número menor de registros, totalizando em 7832 entradas de dado na base.

Um exemplo dos resultados obtidos nesse trabalho pode ser a comparação do tempo de execução de todos os quatro tipos de testes realizados sobre os diferentes tipos de dados, porém com uma quantidade de registro fixada em 5 mil. Para uma melhor visualização, os resultados para esse cenário são mostrados na Figura 5.

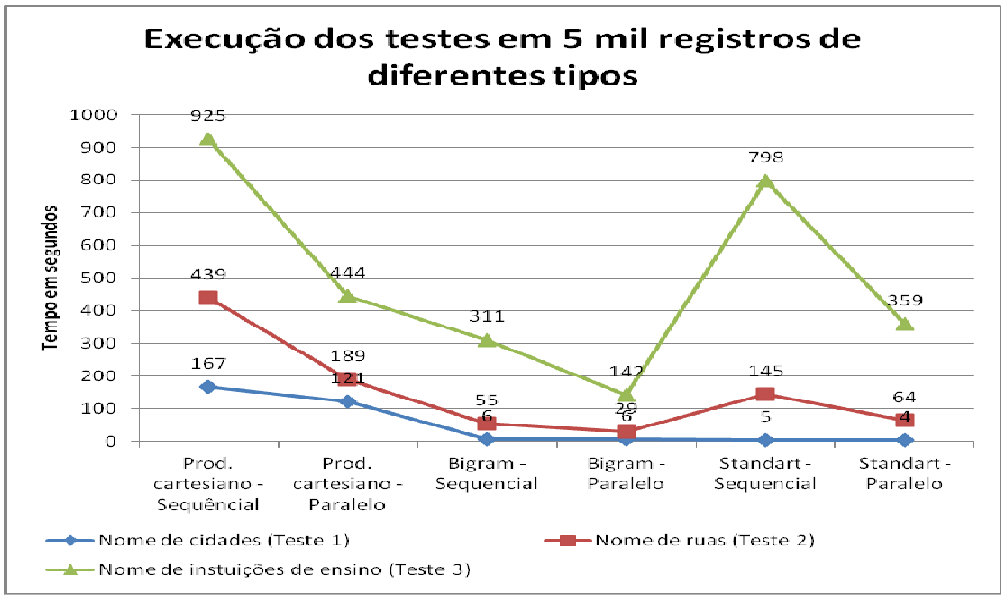
Para uma quantidade de dados iguais, no caso 5 mil registros, é notório que o **teste inst.** foi o que mais levou tempo para processar todas as funções de similaridade. Isso indica que, conforme citado anteriormente, quanto maior a *String* que está sendo avaliada, mais demorada é a execução das funções de similaridade. O **teste cid. e rua** apresentaram características bastante semelhantes, tanto em variação no tempo de execução quanto nos ganhos utilizando a programação paralela. Já o teste 3 é o que apresenta o gráfico mais "desnivelado" demonstrando que foi o teste mais beneficiado quando utilizado o processamento paralelo.

Os maiores ganhos na execução dos algoritmos em paralelo foram registrados nas etapas de teste utilizando produto cartesiano e blocagem utilizando *Standart Index*. Isso comprova que, conforme citado acima, o paralelismo de algoritmos apresenta melhores resultados quando utilizado em um cenário que exija grande processamento

Figura 5 - Gráfico comparativo entre os testes realizados com 5 mil registros

computacional, ou seja, a execuções de muitas tarefas. Como a blocagem utilizando *Bigram Index* se apresentou mais eficiente quanto à geração de pares candidatos mais prováveis, o processamento desnecessário com pares candidatos não semelhantes não é realizado.

8 – Conclusão e trabalhos futuros



Nesse trabalho, foram apresentados alguns resultados que a programação paralela pode trazer para a resolução do problema de deduplicação. Foram mostradas também algumas técnicas de blocagem, o seu funcionamento e a sua implementação. Os resultados demonstrados ao final do trabalho deixam claro o benefício que uma programação

paralela pode trazer quanto ao desempenho da execução de funções de similaridade. A utilização ou não de alguma técnica de blocagem de dados antes de execução das funções também se mostrou favorável à utilização de uma blocagem, mesmo que simples. O tipo de blocagem é algo que deve ser escolhido com enorme critério, pois dependendo do domínio do problema a blocagem acaba por prejudicar a performance da execução das funções de similaridade.

Com uma diferença média no tempo de execução variando entre 75 e 100% nas hipóteses positivas, as funções de similaridade utilizando a estrutura de *Parallel Collections e ParMap* fornecida pela linguagem *Scala*, demonstrou que é possível realizar processamento concorrente de forma mais simples, porém não menos eficaz. Os resultados obtidos foram em todos os casos positivos, tornando a relação entre custo e beneficio de um desenvolvimento mais elaborado para o mesmo problema muito bom. Em questões arquiteturais foi a abordagem mais simples encontrada para se trabalhar com esse tipo de problema.

Por fim, esse trabalho visou esclarecer, corroborando com a literatura, a necessidade de que cada vez mais as aplicações sejam concebidas com o conceito de paralelismo, pois em grande parte do uso, o processador permanece com núcleos ociosos enquanto o desempenho do sistema que está se utilizando não é das melhores. Ainda foi mostrado que a utilização de estruturas paralelas, ou *"programação multi-thread"*, vem sendo simplificada permitindo assim a utilização desses conceitos em diversos cenários.

Entre os trabalhos a serem desenvolvidos no mesmo foco que o P-BSim estão:

- Implementar a blocagem de dados utilizando o mesmo conceito de programação paralela utilizada para a execução das funções de similaridade.
- Realizar experimentos com bases de dados de outros domínios e com um volume de dados maior.
- Tentar solucionar o problema na performance em execuções de funções de similaridade em *Strings* contendo muitos caracteres.
- Aplicar o conceito de programação paralela aqui apresentado em problemas computacionais mais complexos, saindo da vertente de banco de dados.
- Buscar formas mais eficientes porém não mais complexas de executar funções de similaridade entre registros ou qualquer outro processo necessário para solucionar o problema de deduplicação de dados.

9. Referências

Santos, W.; Teixeira, T.; Machado, C.; Meira, Jr. W.; S. Da Silva, A; Ferreira,R.; Guedes, G. **A Scalable Parallel Deduplication Algorithm**, 2007, UFMG

Kawai, H.; Garcia-Molina, H.; Benjelloun, O.; Menestrina, D.; Whang, E.; Gong, H. **P-Swoosh Parallel Algorithm For Generic Entity Resolution**, 2008, Stanford

Friedrich Dorneles, C.; De Matos Galante, R. **Aplicação De Funções De Similaridade E Detecção De Diferenças Em Grandes Volumes De Dados Distribuídos,** 2008, PUC-Rio

Subramaniam, V. **Programming Scala** Tackle Multi-Core Complexity On The Java Virtual Machine. Isbn: 978-1-93435-631-9

F. Gonçalves, C.; Santos, W.; F. D. Flores, L.; S. Vilela, M.; Machado, C.; Meira Jr., W.; Silva, A. **Avaliação De Técnicas Paralelas De Blocagem Para Resolução De Entidades E Deduplicação,** 2008, UFMG

Aleksandar Prokopec, Tiark Rompf, Phil Bagwell, Martin Odersky **A Generic Parallel Collection Framework,** 2010

C. F. Dorneles, C. A. Heuser, V. M. Orengo, A. S. da Silva, E. S. de Moura: **A strategy for allowing meaningful and comparable scores in approximate matching.** Information System, vol 34, n 08, December 2009.