UNIVERSIDADE DE SÃO PAULO

Instituto de Ciências Matemáticas e de Computação

Operadores físicos binários para consultas por similaridade em SGBDR

Luiz Olmes Carvalho

Tese de Doutorado do Programa de Pós-Graduação em Ciências de Computação e Matemática Computacional (PPG-CCMC)

SERVIÇO DE PÓS-GRADUAÇÃO DO ICMC-USP

Data do Depósito:

Assinatura: _____

Luiz Olmes Carvalho

Operadores físicos binários para consultas por similaridade em SGBDR¹

Tese apresentada ao Instituto de Ciências Matemáticas e de Computação – ICMC–USP, como parte dos requisitos para obtenção do título de Doutor em Ciências – Ciências de Computação e Matemática Computacional. *VERSÃO REVISADA*

Área de Concentração: Ciências de Computação e Matemática Computacional

Orientador: Prof. Dr. Caetano Traina Junior

USP – São Carlos
Abril de 2018

¹Trabalho apoiado pela FAPESP (Processo N° 2013/15497-8)

Ficha catalográfica elaborada pela Biblioteca Prof. Achille Bassi
e Seção Técnica de Informática, ICMC/USP,
com os dados inseridos pelo(a) autor(a)

C331o Carvalho, Luiz Olmes
 Operadores físicos binários para consultas por
 similaridade em SGBDR / Luiz Olmes Carvalho;
 orientador Caetano Traina Junior. -- São Carlos,
 2018.
 122 p.

 Tese (Doutorado - Programa de Pós-Graduação em
 Ciências de Computação e Matemática Computacional) --
 Instituto de Ciências Matemáticas e de Computação,
 Universidade de São Paulo, 2018.

 1. junção por similaridade. 2. kNN. 3. quick
 nearest. 4. wide join. 5. operadores relacionais.
 I. Traina Junior, Caetano, orient. II. Título.

Luiz Olmes Carvalho

Physical binary operators for similarity queries in RDBMS²

Doctoral dissertation submitted to the Institute of Mathematics and Computer Sciences – ICMC – USP, in partial fulfillment of the requirements for the degree of the Doctorate Program in Computer Science and Computational Mathematics. *FINAL VERSION*

Concentration Area: Computer Science and Computational Mathematics

Advisor: Prof. Dr. Caetano Traina Junior

USP – São Carlos
April 2018

²This work has been supported by FAPESP (Grant Number 2013/15497-8)

*A todos aqueles que inspiraram
e que nunca lerão.*

*Ao pequeno Luiz Olmes,
quando ele era um garotinho.*

Agradecimentos

Ao *Senhor Deus*, por conceder força e inspiração para a realização deste trabalho.

A meus pais, *Maria Lades da Silva Carvalho* e *Ivo Guimarães de Carvalho*, por todo o amor, apoio e compreensão durante toda a vida.

Ao *Prof. Dr. Caetano Traina Jr.*, por sua orientação sólida, precisa e segura, pelo aprendizado e amadurecimento de diversos conceitos, pela disponibilidade e empenho manifestados. À *Profa. Dra. Agma Juci Machado Traina*, por seu incentivo, atenção, apoio e encorajamento.

Aos amigos para todo o sempre, *Christian César Bones* e *Clélia Junko Kinzú Dimário* (“*Lia*”), por se tornarem parte da minha família e me acolherem na de vocês. Obrigado por estarem sempre presentes ao longo desta caminhada. Sinto falta de vocês por perto!

Aos companheiros cujos caminhos se cruzaram ao meu diante de um ideal em comum, e, por todos os momentos compartilhados, a saudade há de ficar. Em especial, *Lúcio Fernandes Dutra Santos*, *Marcos Vinícius Naves Bedo* e *Willian Dener de Oliveira*; sempre em ordem alfabética, como as coisas devem ser.

A uma das pessoas mais fantásticas com quem tive o privilégio de conviver, o amigo *Enzo Seraphim*.

Àqueles cujo primor e esmero despontam como fagulhas de ânimo nos mais turvos dias, inspirando e motivando, *Fernando Aparecido de Assis* e *Kleber Mazione Lima Ferreira*.

À *Fundação de Amparo à Pesquisa do Estado de São Paulo - FAPESP*, por acreditar e apoiar este doutorado sob o Processo N° 2013/15497-8.

Aos *professores do Instituto de Ciências Matemáticas e de Computação*, pelo conhecimento compartilhado, e à *Universidade de São Paulo*, pelo uso de suas instalações, recursos e equipamentos.

Aos membros da *Banca Examinadora*, pelos valiosos comentários, sugestões de melhorias e disponibilidade para apreciação deste trabalho.

A todos que, embora não citados, contribuíram direta ou indiretamente para a realização deste trabalho, expresso meu sincero reconhecimento e agradecimento. Que a ausência de seus nomes nunca signifique o esquecimento.

*Todos esses que aí estão
Atravancando o meu caminho,
Eles passarão...
Eu passarinho!*

MÁRIO QUINTANA

Resumo

Carvalho, L. O. **Operadores físicos binários para consultas por similaridade em SGBDR**. 2018. 122 p. Tese (Doutorado em Ciências de Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos - SP, 2018.

O operador de Junção é um operador importante da Álgebra Relacional que combina os pares de tuplas que atendem a uma dada condição de comparação entre os valores dos atributos de duas relações. Quando a comparação avalia a similaridade entre pares de valores, o operador é chamado “Junção por Similaridade”. Esse operador tem aplicações em diversos contextos, tais como o suporte de tarefas de mineração e análise de dados em geral, e a detecção de quase-duplicatas, limpeza de dados e casamento de cadeias de caracteres em especial. Dentre os operadores de junção por similaridade existentes, a Junção por Abrangência (*range join*) é a mais explorada na literatura. Contudo, ela apresenta limitações, tal como a dificuldade para se encontrar um limiar de similaridade adequado. Nesse contexto, a Junção por *k*-vizinhos mais próximos” (*k-nearest neighbor join* – *kNN join*) é considerada mais intuitiva, e portanto mais útil que o *range join*. Entretanto, executar um *kNN join* é computacionalmente mais caro, o que demanda por abordagens baseadas na técnica de laço aninhado, e as técnicas existentes para a otimização do algoritmo são restritas a um domínio de dados em particular. Visando agilizar e generalizar a execução do *kNN join*, a primeira contribuição desta tese foi o desenvolvimento do algoritmo *QuickNearest*, baseado na técnica de divisão e conquista, que é independente do domínio dos dados, independente da função de distância utilizada, e que computa *kNN joins* de maneira muito eficiente. Os experimentos realizados apontam que o *QuickNearest* chega a ser 4 ordens de magnitude mais rápido que os métodos atuais. Além disso, o uso de operadores de junção por similaridade em ambientes relacionais é problemático, principalmente por dois motivos: (i) em geral o resultado tem cardinalidade muito maior do que o realmente necessário ou esperado pela maioria das aplicações de análise de dados; e (ii) as consultas que os utilizam envolvem também operações de ordenação, embora a ordem seja um conceito não associado à teoria relacional. A segunda contribuição da tese aborda esses dois problemas, tratando os operadores de junção por similaridade existentes como casos particulares de um conjunto mais amplo de operadores binários, para o qual foi definido o conceito de *Wide-joins*. Os operadores *wide-joins* recuperam os pares mais similares em geral e incorporam a ordenação como uma operação interna ao processamento, de forma compatível com a teoria relacional e que permite restringir a cardinalidade dos resultados a tuplas de maior interesse para as aplicações. Os experimentos realizados mostram que os *wide-joins* são rápidos o suficiente para serem usados em aplicações reais, retornam resultados de qualidade melhor do que os métodos concorrentes e são mais adequados para execução num ambiente relacional do que os operadores de junção por similaridade tradicionais.

Palavras-chave: Junção por similaridade, *kNN*, *QuickNearest*, *wide-join*, operadores relacionais.

Abstract

Carvalho, L. O. **Physical binary operators for similarity queries in RDBMS**. 2018. 122 p. Thesis (Doutorado em Ciências de Computação e Matemática Computacional) - Instituto de Ciências Matemáticas e de Computação, Universidade de São Paulo, São Carlos - SP, 2018.

Joins are important Relational Algebra operators. They pair tuples from two relations that meet a given comparison condition between the attribute values. When the evaluation compares the similarity among the values, the operator is called a “Similarity Join”. This operator has application to a variety of contexts, such as supporting data mining tasks and data analysis in general, and near-duplicate detection, data cleaning and string matching in particular. Among the existing types of similarity joins, the *range join* is the most explored one in the literature. However, it has several shortcomings, such as the difficulty to find adequate similarity thresholds. In such context, the *k-nearest neighbors join* (*kNN join*) is considered more intuitive, and therefore more useful than the range join. However, the *kNN join* execution is computationally well more expensive, thus demanding implementations either based on nested loop techniques, which are generic, or on optimizing techniques but that are specific data given domains. In order to accelerate and generalize *kNN join* execution, the first contribution of this thesis was the development of the *QuickNearest* algorithm, based on the divide and conquest approach that is independent of the data domain, independent of the distance function used, and that computes *kNN joins* very efficiently. Experiments performed with the *QuickNearest* algorithm show that it is up to four orders of magnitude faster than current methods. Nevertheless, using similarity join operators in relational environments remains generally troublesome, due to two main reasons: (i) the result often has a cardinality much larger than what is actually needed or expected by most of the data analysis applications; and (ii) queries that use them almost always also require sorting operations, but order concept is not present in the relational theory. The second contribution of the thesis addresses these two problems through the definition of the concept of *Wide-joins*, which turns the existing similarity join operators just as particular cases of a more powerful set of binary operators. A *wide-join* operator retrieves the pairs most similar in general and already incorporates ordering as an internal operation to its processing, what makes it fully compatible with the relational theory. The concept also provides powerful ways to restrict the result cardinality just to tuples really meaningful for the applications. In fact, the experiments have also shown that *wide-joins* are fast enough to be useful for real applications, they return results of better quality than competing methods, and are more suitable for execution in a relational environment than the traditional similarity join operators.

Keywords: Similarity join, *kNN*, *QuickNearest*, *wide-join*, relational operators.

Lista de Figuras

2.1	Formato da região delimitada pela função Minkowski	12
2.2	Operadores de seleção por similaridade	18
2.3	Operadores de junção por similaridade	21
2.4	Operadores lógicos encapsulados em operadores de acesso físico	28
3.1	Particionamento do espaço de busca por hiperplano generalizado	40
3.2	Particionamento do espaço gerado pelo método <code>joinWindows()</code>	43
3.3	Definição da <i>window distance</i>	49
3.4	Particionamento do espaço de busca por bolas métricas	52
3.5	Árvore de recursão	58
3.6	Avaliação da escalabilidade – conjunto GEONet/USGS	61
3.7	Avaliação da dimensionalidade – conjunto Nasa	63
3.8	Avaliação da dimensionalidade – conjunto Aloï	64
3.9	Avaliação da dimensionalidade – conjunto Faces	65
3.10	Avaliação da distribuição de dados – conjuntos sintéticos	67
3.11	Avaliação de dados puramente métricos – conjunto UniProt	68
3.12	Variação de parâmetros: k e H	70
3.13	Variação de parâmetros: k e H (cont.)	71
4.1	<i>Wide-join</i> com predicados distintos	81
4.2	Avaliação da escalabilidade do <i>wide-join</i>	86
4.3	Estratégia baseada em pivôs para realizar podas no espaço de busca	89
4.4	Avaliação de dados puramente métricos – conjunto Fire	92

4.5	Quase duplicatas obtidas pelos algoritmos de junção	93
4.6	Avaliação da escalabilidade do <i>self range wide-join</i> – conjunto Aloï	94
A.1	Limite inferior da distância de um objeto ao hiperplano	116

Lista de Tabelas

2.1	Conjunção	14
2.2	Disjunção	14
2.3	Negação	14
2.4	Operadores de seleção e junção por similaridade	31
4.1	Configuração paramétrica dos experimentos	84

Abreviaturas e Siglas

ACMd	– <i>Adaptive Clusters with k-Medoids</i>
ACMe	– <i>Adaptive Clusters with k-Means</i>
AlgKNN	– <i>algebraic k-nearest neighbor wide-join</i>
AlgRng	– <i>algebraic range wide-join</i>
EGO	– <i>Epsilon Grid Order</i>
FALSE	– valor lógico falso
GIS	– <i>Geographical Information System</i>
I/O	– operação de entrada/saída (input/output)
IDX	– algoritmo NLJ com relação interna de junção indexada
KNN	– <i>k-nearest neighbor wide-join</i>
kNN	– <i>k-nearest neighbor</i>
LTC	– <i>List of Twin Clusters</i>
NLJ	– <i>Nested-Loop Join</i>
NULL	– valor nulo de atributo
NotAlgKNN	– <i>algebraic not-the-k-nearest neighbor wide-join</i>
NotAlgRng	– <i>algebraic not-in-range wide-join</i>
NotKNN	– <i>not-the-k-nearest neighbor wide-join</i>
NotRng	– <i>not-in-range wide-join</i>
OLAP	– <i>On-line Analytical Processing</i>
OLTP	– <i>On-line Transaction Processing</i>
PACS	– <i>Picture Archiving and Communication Systems</i>
PSWJ	– <i>Pivot-based Self Wide-Join</i>
QNB	– algoritmo <i>QuickNearest</i> com particionamento por bolas métricas
QNH	– algoritmo <i>QuickNearest</i> com particionamento por hiperplano generalizado
RI	– Relação de Identidade
RO	– Relação de Ordem
ROL	– Relação de Ordem Linear
ROLE	– Relação de Ordem Linear Estrita

RS	–	Relação de Similaridade
Rng	–	<i>range wide-join</i>
SGBD(R)	–	Sistema de Gerenciamento de Bases de Dados (Relacional)
SQL	–	<i>Structured Query Language</i>
SimKNN	–	<i>similarity k-nearest neighbor join</i>
SimRng	–	<i>similarity range join</i>
TRUE	–	valor lógico verdadeiro
UNKNOWN	–	valor lógico desconhecido

Símbolos

φ	– conectivo lógico
θ	– operador de comparação
θ_s	– operador de comparação por similaridade
ρ, ω	– frações de conjuntos
τ	– termo de predicado
ξ	– limiar de similaridade
$\mathbb{A}_i$	– domínio de atributo em geral (escalar ou complexo)
A_i	– atributo (escalar ou complexo)
Ard	– operador de comparação do <i>join around</i>
ID	– domínio de dados
d	– função de distância
d_k	– k -ésima distância
$\mathbb{E}_i$	– domínio de atributo escalar
E_i	– atributo escalar
e_i	– elemento de conjunto
$\mathcal{F}$	– função de ordenação
G, G^w, L, L^w	– partições e janelas do <i>Quickjoin</i>
H	– limiar de sublimação
k, κ_f	– constante numérica
kCN	– operador de comparação <i>k-closest neighbor</i>
kNN	– operador de comparação <i>k-nearest neighbor</i>
L_p	– função Minkowski
ℓ	– último nível da recursão
$\mathcal{M}$	– espaço métrico
m, μ, n	– tamanho de conjunto de dados
$\mathbb{N}^*$	– conjunto dos números naturais sem o zero
NNJ	– <i>nearest neighbor join</i>
NNS	– <i>nearest neighbor selection</i>

ord	– atributo escalar que armazena a classificação ordinal no <i>wide-join</i>
p	– predicado de comparação
p_i	– elemento de referência (pivô)
Q	– fila de prioridade (<i>queue</i>)
q	– elemento de consulta
R, S, T, U	– relações
$\mathbb{R}_+$	– conjunto dos números reais não negativos
Rng	– operador de comparação <i>similarity range</i>
r, r_i	– tuplas com elementos complexos
$\mathcal{S}$	– domínio de dados complexos
$\mathcal{S}_i$	– atributo complexo
s, s', s_i, s_j	– tuplas com elementos complexos
s_q	– centro de consulta complexo
t	– tupla
v_i	– característica i do vetor de características v
x, y, z	– elementos de conjuntos
$\mathbb{Z}_+^*$	– conjunto dos números inteiros positivos

Sumário

Lista de Figuras	xvi
Lista de Tabelas	xvii
Abreviaturas e Siglas	xix
Símbolos	xxi
1 Introdução	1
1.1 Motivação e problemática	3
1.2 Teses	6
1.3 Objetivos	8
1.4 Estrutura do documento	8
2 Consultas por similaridade	9
2.1 Modelo de similaridade	9
2.2 Operadores de comparação por similaridade	12
2.3 Seleção por similaridade	16
2.4 Junção por similaridade	18
2.5 Abordagens de junção por similaridade	20
2.6 Operadores físicos para junção	26
2.7 Considerações finais	30
3 QuickNearest	33
3.1 Definição do operador <i>k-nearest neighbor join</i>	33

3.2	A estrutura do conjunto-resposta	34
3.3	O algoritmo <i>QuickNearest</i>	36
3.3.1	Resposta aproximada $\times$ resposta exata	39
3.3.2	O limiar de sublimação H	44
3.4	Estratégia de particionamento por hiperplano generalizado	46
3.5	Estratégia de particionamento por bolas	51
3.6	Análise do algoritmo <i>QuickNearest</i>	56
3.7	Experimentos	59
3.7.1	Escalabilidade	60
3.7.2	Dimensionalidade dos dados	63
3.7.3	Distribuição de dados	66
3.7.4	Dados adimensionais	68
3.7.5	Variação paramétrica	69
3.8	Considerações finais	72
4	Wide-join	75
4.1	Derivação do operador <i>wide-join</i>	76
4.2	Definição do operador <i>wide-join</i>	77
4.3	Predicados do <i>wide-join</i>	79
4.3.1	Predicados de termo único	79
4.3.2	Negação de predicados de termo único	80
4.3.3	Predicados de termos múltiplos	83
4.4	Experimentos	84
4.5	O operador <i>self wide-join</i>	87
4.5.1	Um algoritmo para <i>self range wide-joins</i>	88
4.5.2	Aplicação do <i>self wide-join</i> para detecção de quase-duplicatas	90
4.5.3	Avaliação do operador <i>self range wide-join</i>	91
4.6	Propriedades algébricas envolvendo <i>wide-joins</i>	94
4.7	Considerações finais	97

5 Conclusão	99
5.1 Contribuições	100
5.2 Dificuldades encontradas	102
5.3 Publicações	104
Referências Bibliográficas	107
A Distância para um hiperplano generalizado	115
B Demonstração das propriedades algébricas envolvendo <i>wide-joins</i>	117

Introdução

Um *Sistema de Gerenciamento de Bases de Dados (Relacional)* (SGBD(R)) é um *software* de uso geral que facilita o processo de definição, coleta, manipulação e compartilhamento e análise de dados entre diversos usuários e aplicações [32]. Além disso, um SGBD fornece acesso a dados de forma estruturada, gerenciável e auditável, seguindo uma estrutura-modelo do dado armazenado.

Inicialmente os SGBDs foram projetados visando o armazenamento e a recuperação de grandes coleções de dados simples, de modo que sempre é possível responder as consultas de forma exata e geralmente eficiente [34]. A maioria dos SGBDs atuais baseia-se na tecnologia relacional, que começou a ser desenvolvida no final da década de 1960 [27], sendo que esses sistemas foram estruturados para manipular números, datas e pequenas cadeias de caracteres, chamados *dados escalares* [28].

A manipulação de dados em domínios escalares é realizada usando *operadores de comparação*. Os operadores de comparação por *igualdade*, usualmente representados pelos símbolos $=$ e $\neq$, introduzem a Relação de Identidade (RI) entre os elementos envolvidos na comparação. A princípio, estes operadores podem ser aplicados a quaisquer tipos de dados escalares, uma vez que é sempre possível arbitrar sobre a igualdade de dois elementos. Os operadores de comparação baseados em Relação de Ordem (RO), denotados por $<$, $\leq$, $\geq$ e $>$, avaliam a *precedência* de um elemento sobre outro. Para isso, esses operadores fazem uso das propriedades conhecidas como Relação de Ordem Linear (ou Total) e Relação de Ordem Parcial, que podem ser definidas sobre os domínios de dados escalares.

Esses seis operadores são nativamente suportados pelas linguagens de programação, e também naturalmente oferecidos pelos SGBDs. Eles são conhecidos como os comparadores “*big six*” do modelo relacional e da linguagem SQL [63, 28, 17]. Entretanto, eles não são os únicos operadores de comparação disponíveis: outros operadores também podem ser aplicados, como, por exemplo, a continência de uma subcadeia de caracteres em outra.

Uma vez que os operadores de comparação por RI e RO são aplicáveis a todos os tipos de dados escalares, estas propriedades possibilitaram o desenvolvimento de algoritmos de indexação e

persistência bastante eficientes [32, 28, 15]. Contudo, com a evolução das aplicações que utilizam SGBDRs, vem se tornando necessário organizar, armazenar e recuperar tipos de dados não escalares, como imagens e dados georreferenciados, para os quais os comparadores *big six* são pouco úteis, ou até mesmo inadequados [24, 34]. Por exemplo, em Sistemas de Informações Geográficas (GIS - *Geographical Information System*), as consultas mais frequentes são baseadas em relações espaciais entre os elementos, tais como Relações Cardeais (por exemplo, elementos que estão “ao norte” de outro), Relações Topológicas (interceptação ou sobreposição de regiões) e Relações de Distância. Dessa maneira, os operadores de busca típicos para essas consultas devem manipular propriedades espaciais derivadas dessas relações [60].

Operadores baseados em RI são inadequados para diversos domínios de dados (por exemplo, coleções de imagens), visto que é raro existirem dois elementos exatamente iguais. Já os operadores baseados em relações de ordem não são genericamente aplicáveis, pois o domínio em que estes dados estão definidos geralmente não segue estas relações. Por exemplo, não é possível ordenar imagens, a menos que seja utilizada alguma informação externa associada ao dado, como o nome ou data da imagem; mas essa ordenação não reflete características visuais, que deveriam ser as consideradas para a recuperação, ou seja, para recuperar as imagens baseando-se em seu conteúdo [33, 24]. Dados não escalares (e mesmo dados escalares), em geral podem ser comparados através da noção de *similaridade*, usando uma *função de distância* para mensurar o quanto dois elementos são similares entre si, permitindo sua comparação e recuperação [93]. Dados que não são comparados diretamente, mas por meio de uma função de distância são genericamente chamados de *dados complexos*. Exemplos frequentes de dados complexos são os dados multimídia (imagens, áudio, vídeo), séries temporais, sequências genéticas, grafos e informações georreferenciadas [34, 17].

De modo análogo aos dados escalares, que são comparados por operadores embasados nas propriedades de RI e RO, a comparação de dados complexos é baseada em propriedades de Relações de Similaridade (RS). Existem dois operadores de comparação por similaridade fundamentais [7, 17]. O primeiro deles, chamado “*comparação por abrangência*” (*similarity range*) faz uso de um limiar de similaridade e avalia se a medida de similaridade entre um elemento qualquer e um elemento de consulta (centro de consulta) está, ou não, inclusa nesse limiar. O segundo operador, chamado “*comparação por vizinhos mais próximos*” (*k-nearest neighbor – kNN*) é baseado numa quantidade de elementos que são os mais similares a um centro de consulta, expressando, assim, comparações de similaridade por vizinhança [7, 19].

Para que algoritmos eficientes possam ser construídos para executar consultas que usem esses dois operadores, é necessário considerar as propriedades que os dados atendem. Assim, consultas baseadas em comparações por similaridade sobre dados complexos normalmente assumem que os dados estão definidos em um *espaço métrico* [57]. Assim, a Teoria dos Espaços Métricos constitui o formalismo matemático que embasa as consultas por similaridade.

A recuperação de dados nos SGBDRs é realizada através de *operadores de consulta* (também chamados *operadores de busca*), como projeção, união, seleção e junção [28]. Notadamente, os operadores de seleção e junção são aqueles que avaliam predicados expressos explicitamente, ou seja, eles permitem indicar quais são as comparações de valores dos atributos das tuplas que devem ser executadas. Devido à necessidade de armazenar e recuperar dados complexos em SGBDs, os ope-

radores de busca baseados em operadores de comparação por similaridade tornam-se atrativos, principalmente para a recuperação por conteúdo [72, 7, 16, 75, 83, 77, 61, 26].

Um *operador de seleção* é um operador de aridade 1, utilizado para “filtrar” as tuplas de uma relação que atendem a um predicado. Quando o predicado expressa comparações por similaridade, o operador de consulta resultante é chamado de operador de *seleção por similaridade*. Existem duas instâncias do operador de seleção por similaridade: a seleção por abrangência, que faz uso do operador de comparação por abrangência, e a seleção por vizinhança, empregando o operador de comparação por k vizinhos mais próximos (kNN) [7, 17].

Por sua vez, um *operador de junção* é um operador binário que “emparelha” os pares de tuplas de duas relações que atendem ao respectivo predicado de junção. De modo análogo ao operador de seleção, o emprego de operadores de comparação por similaridade na condição de junção resulta em operadores de *junção por similaridade*. Existem três operadores de junção por similaridade descritos com mais frequência na literatura: junção por abrangência (*range join*), junção por vizinhança (*k-nearest neighbor join*) e junção por proximidade (*k-closest neighbor join*, também chamado de *k-closest pair join* ou *k-distance join*) [14, 7, 75, 19].

O operador *range join* faz uso de um limiar de similaridade para recuperar os pares de elementos que estão dentro dessa distância predefinida. Este operador de junção por similaridade é o que aparece com mais frequência na literatura, tanto que muitos trabalhos referem-se a ele apenas como “o operador de junção por similaridade” (*the similarity join database operator*) [49, 76, 77, 69].

Os outros dois operadores de junção por similaridade baseiam-se na recuperação de uma quantidade k de pares de elementos. Um *k-nearest neighbor join* emprega o operador de comparação por similaridade kNN a fim de recuperar, para cada elemento da primeira relação, os k elementos mais similares da segunda relação. Por sua vez, o *k-closest neighbor join* seleciona os k pares de elementos mais similares em todo o conjunto. Para recuperar os k elementos mais similares em um contexto geral, um *k-closest neighbor join* executa, internamente, outras operações intermediárias, incluindo uma ordenação. Os operadores de seleção e junção por similaridade são detalhados no capítulo 2.

1.1 Motivação e problemática

A junção por similaridade é uma das operações mais utilizadas no processamento de consultas, seja para dados tradicionais ou complexos, e tem sido extensivamente investigada [49, 84, 74, 85, 55, 75, 76, 17, 77, 19, 20, 61, 69, 26, 25, 21].

Operadores de junção por similaridade são aplicáveis a diversos contextos, desde aqueles que demandam processamento interno a um SGBD, e, conseqüentemente, sua integração ao gerenciador [77, 61], até técnicas externas a SGBDs que empregam operadores de bancos de dados para responder consultas por similaridade [23, 49, 21]. Como exemplo dos diversos tipos de aplicações que fazem uso de operadores de junção por similaridade, destacam-se:

- **Limpeza de dados** (*data cleaning*) [23, 25]: operadores como o *range join* são essenciais para a tarefa de *data cleaning*. Por exemplo: em um *data warehouse* de vendas, erros de digitação

ou falta de padronização nos cadastros podem fazer com que os nomes dos produtos ou dos clientes nos registros de venda não correspondam exatamente àqueles armazenados no catálogo geral ou nos registros dos consumidores, resultando em erros no processamento de consultas. O operador de junção por similaridade pode ser aplicado para reduzir esses erros, onde funções de distância entre caracteres permitem mensurar a similaridade entre os nomes dos produtos ou clientes.

- **Sistemas de recomendação em tempo-real** [86, 5]: através do mapeamento de produtos e preferências de usuários para o mesmo espaço de busca, uma técnica frequentemente empregada é recomendar aos usuários aqueles itens que estão entre os k mais similares a uma busca realizada ou relacionados a um item anteriormente comprado. Uma vez que vários usuários *online* devem ser simultaneamente respondidos, a tarefa de processar a lista de recomendação para todos eles é traduzida por um *k-nearest neighbor join*.
- **Suporte a algoritmos de aprendizado de máquina** [14]: algoritmos para diversas tarefas da área de *machine learning* tais como agrupamento, classificação, detecção de anomalias, entre outros, geralmente necessitam que os elementos de um conjunto sejam combinados com os representantes mais similares em outro conjunto. Por exemplo, um classificador k NN rotula elementos desconhecidos com base em seus vizinhos mais próximos. Se houverem vários elementos não rotulados, a operação de classificação corresponde a um *k-nearest neighbor join*.
- **Sistemas de informações geográficas** [38]: aplicações envolvendo informações georreferenciadas podem se beneficiar de algoritmos de junção por similaridade. Por exemplo, considerando um agendamento eficiente de instalações turísticas, o operador *k-closest neighbor join* permite recuperar os k pares mais próximos de locais culturais e hotéis.
- **Detecção de quase-duplicatas** (*near-duplicate detection*) [85, 18, 21]: a detecção de objetos quase duplicados é importante em diversos contextos. Por exemplo, um sistema de arquivamento de informações médicas como o PACS (*Picture Archiving and Communication Systems*), é capaz de armazenar milhares de imagens de exames. Objetivando apoiar o processo de tomada de decisão, a recuperação de imagens similares às imagens do paciente atual pode auxiliar os médicos na prescrição de um tratamento similar ou na escolha da forma de intervenção. Ambos os operadores *range join* e *k-nearest neighbor join* auxiliam a responder a essa consulta.
- **Processamento de streams** [87, 80, 65]: diversas tarefas relacionadas ao processamento de *data streams* são implementadas usando operadores de junção por similaridade como o *range join* e o *k-nearest neighbor join*. Por exemplo, técnicas para descoberta de padrões de interesse em mineração de *streams* e para detecção de tendências em redes sociais podem ser implementadas usando um *k-nearest neighbor join* como operador base do processamento.
- **Problemas de combinação de strings** (*string matching*) [68, 90]: fazendo uso do operador *range join*, estas técnicas são voltadas para identificar similaridade entre cadeias de caracteres (*strings*). Elas podem ser aplicadas em diversas tarefas, como, por exemplo, verificação gramatical, integração de dados e para processar dados representados através de *strings*, tais como sequências genéticas e sequências proteicas.

Apesar de sua enorme versatilidade e aplicabilidade, os três operadores de junção por similaridade não são livres de problemas ou limitações, o que justifica sua exploração extensiva pela literatura atual de Bases de Dados, e é o foco desta tese.

A **primeira limitação** dos operadores de junção por similaridade é a operação apresentar complexidade temporal quadrática no tamanho n do conjunto de dados, o que significa que o desempenho dos operadores é proporcional ao quadrado do tamanho da entrada. Um algoritmo de força bruta com custo $O(n^2)$ é o algoritmo mais eficiente para se computar uma junção por similaridade cuja resposta deve ser *exata* (não aproximada) [65]. De fato, até onde o conhecimento do autor permite pontuar, o algoritmo de laço aninhado (*nested-loop*) é a única técnica existente que é capaz de computar qualquer um dos três tipos de junção por similaridade, tendo qualquer tipo de dado (multidimensional, adimensional ou puramente métrico) como entrada, e usando qualquer função de distância, seja ela métrica ou não-métrica. Portanto, uma junção por similaridade é uma operação computacionalmente cara de ser executada.

Devido ao alto custo computacional de uma junção por similaridade, diversas abordagens têm sido propostas visando acelerar o processamento deste operador. Estas propostas são divididas, basicamente, em técnicas indexadas [31, 67, 38, 26] e não-indexadas [13, 14, 37]. A **segunda limitação** destes operadores reside no fato dos avanços alcançados não serem genericamente aplicáveis. A maioria das técnicas desenvolvidas são exclusivas para o operador *range join*. Além disso, elas não são capazes de processar dados de qualquer tipo e apresentam limitações quanto às funções de distância que podem usar. Outras abordagens especialmente desenvolvidas para otimizar *k-nearest neighbor joins* computam respostas aproximadas [36, 37], onde pares relevantes são substituídos por outros menos similares.

O **terceiro problema** do operador de junção por similaridade é não permitir que estruturas de índice sejam amplamente empregadas em seu processamento. Em ambientes transacionais (OLTP - *On-line Transaction Processing*), o plano de execução de consulta gerado pelo otimizador do SGBD(R) primeiramente faz uso das estruturas de índices existentes sob a forma de chaves para filtrar tuplas, para só então combinar as tuplas vindas de mais de uma relação usando o operador de junção [32, 71]. Devido a heterogeneidade dos resultados parciais da primeira filtragem para cada consulta, torna-se difícil estabelecer um índice para junção, a menos que a consulta seja muito frequente, o que é característica de aplicações analíticas. Portanto, abordagens indexadas para resolver o operador de junção são melhor empregadas em ambientes analíticos (OLAP - *On-line Analytical Processing*) do que em sistemas transacionais [15].

Como **quarta restrição** dos operadores de junção, tem-se as diferenças de processamento de predicados. Predicados que compõe as junções *range* e *k-closest neighbor* são intercambiáveis com predicados compostos pelos operadores “*big six*” [75], o que permite que eles sejam processados em qualquer ordem, de acordo com o plano de execução de consultas mais conveniente. Já um predicado do tipo *k-nearest neighbor* não preserva as propriedades da comutatividade, associatividade e distributividade [34, 3, 75] sobre os demais operadores, devendo ser processados na ordem em que aparecem, limitando as opções de otimização de consulta.

Em uma consulta por similaridade, quase sempre o interesse está nos elementos mais similares, seja porque esses são os elementos mais úteis ao usuário ou os mais relevantes para a aplicação

que executa a consulta [62, 38, 18]. Apesar dos operadores de junção por similaridade realizarem a recuperação dos objetos mais similares, o **quinto problema** destes operadores é que, em geral, eles retornam um conjunto resposta de cardinalidade elevada. Por exemplo, dados dois conjuntos, ou duas relações, R e S , um *k-nearest neighbor join* recupera $k \cdot |R|$ pares de elementos, ou seja, uma quantidade de elementos k vezes maior que o tamanho de um dos conjuntos de entrada. Como consequência de se obter um conjunto resposta de alta cardinalidade, tem-se que o usuário pode ser induzido a acreditar que sua consulta não foi respondida, haja visto que a resposta desejada encontra-se imersa entre os elementos de menor similaridade, fazendo com que o usuário reescreva e reexecute a consulta, ou acabe por desistir dela [70].

Nesse cenário, o operador *k-closest neighbor join* apresenta aplicabilidade bem maior que os dois outros operadores de junção por similaridade, pois retorna respostas com menor cardinalidade e de maior interesse aos usuários e/ou aplicação. De fato, este operador devolve exatamente k pares de elementos e esses pares correspondem aos elementos mais similares em geral. Entretanto, a busca pelos pares mais similares entre todo o conjunto necessita de operações internas auxiliares. Por exemplo, para se obter a resposta final, é necessário ordenar todo o conjunto-resposta usando o valor da dissimilaridade entre os pares como atributo de ranqueamento. Além disso, é necessário estabelecer uma forma de representar e incluir esse atributo de classificação nas tuplas.

A necessidade de operações intermediárias ao processamento constitui o **sexto problema** dos operadores de junção por similaridade. Uma junção por similaridade é definida como uma seleção sobre o resultado de um produto Cartesiano, onde a seleção emprega um predicado por similaridade dos tipos *range* ou *kNN*. A presença de operadores internos adicionais além destes operadores fundamentais da álgebra no processamento de um *k-closest neighbor join* faz com este último operador não siga a definição relacional do operador de junção. Portanto, um *k-closest neighbor join* pode ser visto como um tipo de operador binário cuja definição engloba o conceito de junção [19].

1.2 Teses

Algumas das restrições dos operadores de junção por similaridade são passíveis de pesquisas e propostas de otimizações, enquanto que outras, ao menos nos moldes da computação atual, ainda não podem ser trabalhadas.

As primeira e quarta limitações apresentadas na seção 1.1 não são solucionáveis. Um operador de junção por similaridade compara todos os elementos do primeiro conjunto de dados com todos os elementos do segundo conjunto de dados, e a comparação de dois conjuntos distintos é uma operação inerentemente quadrática. Já na quarta limitação, a falta das propriedades de comutatividade, associatividade e distributividade se apresenta como característica intrínseca do operador *k-nearest neighbor join*. Da mesma forma que uma adição é comutativa ($a + b = b + a$) e uma subtração não é ($a - b \neq b - a$), junções *range* e *k-closest neighbor* permitem explorar estas propriedades, enquanto que um *k-nearest neighbor join* não permite.

Os segundo e terceiro problemas relatados restringem a aplicabilidade do operador de junção. Devido às operações executadas antes do processamento de uma junção, métodos de indexação tornam-se pouco úteis. Além disso, a pouca existência de trabalhos enfocando além do operador *range join* [75] representa outra restrição, considerando, ainda, que eles são dependentes de domínios de dados e/ou funções de distância específicas.

Entretanto, algumas técnicas de consulta por similaridade [81, 26] se baseiam em propriedades do espaço métrico a fim de reduzir a quantidade de comparações realizadas por operadores de seleção por similaridade, ou mesmo *range join* [49, 67]. Esses métodos excluem do espaço de busca os elementos que garantidamente não satisfazem as condições para serem incluídos como resposta, possibilitando que, ainda na falta das propriedades de comutatividade, associatividade e distributividade, consultas do tipo *kNN* obtenham maior desempenho. Nesse contexto, a principal tese formulada para este trabalho é:

Tese 1. *A investigação de propriedades do espaço métrico, de técnicas de particionamento de elementos e de propriedades associadas a funções de distâncias possibilita construir algoritmos independentes de domínios para otimizar o desempenho do operador ‘k-nearest neighbor join’.*

O quinto problema identificado na seção 1.1 permite pontuar que os operadores de junção por similaridade se restringem a tarefas de preprocessamento de dados [19], quase sempre atuando como operadores intermediários da consulta, uma vez que a alta cardinalidade do conjunto-resposta normalmente necessita de filtragens adicionais [23, 85]. Por sua vez, o sexto problema mostra que o operador *k-closest neighbor join* viola a definição algébrica de junção (por incluir operações extras além de seleção e produto Cartesiano) e, também, viola a Álgebra Relacional (por ser dependente de aspectos inexistentes na álgebra, como a questão da ordenação).

Através da análise de todas as operações executadas para computar um *k-closest neighbor join*, pode-se perceber que a “violação” da definição algébrica de um operador de junção presente no referido operador, se aplicada aos demais operadores de junção por similaridade, permite obter resultados mais expressivos e de menor cardinalidade, aumentando sua usabilidade e reduzindo a necessidade de processamento adicional após a junção. Nesse contexto, as quinta e sexta limitações dos operadores de junção culminam na segunda tese sustentada pelo presente trabalho, que é enunciada da seguinte maneira:

Tese 2. *A identificação e extensão das propriedades do operador “k-closest neighbor join” permite definir uma nova classe de operadores de consulta por similaridade binários que inclui a exploração de conceitos adicionais ao processamento, como a ordem, internamente a um operador por similaridade.*

A investigação conduzida por este trabalho concentra-se nas duas teses apresentadas, e desponha nos objetivos descritos na próxima seção 1.3.

1.3 Objetivos

Esta tese está centrada em dois objetivos fundamentais:

1. *Desenvolver algoritmos que otimizem o desempenho do operador “ k -nearest neighbor join”*: o algoritmo de laço aninhado é a única técnica existente para computar k -nearest neighbor joins em geral. Através da exploração de propriedades do espaço métrico e de técnicas computacionais como divisão e conquista, chega-se ao desenvolvimento de algoritmos eficientes para uma junção do tipo k -nearest neighbor e à otimização do operador.
2. *Definir uma nova classe de operadores binários por similaridade que englobem o conceito de junção e ordenação*: os operadores *range join* e *k-nearest neighbor join* são, de fato, junções, mas o operador *k-closest neighbor join* se encaixa em uma classe distinta, dependente, entre outros, de ordenação. Este operador é o primeiro elemento dessa nova classe de operadores binários e ponto de partida para a identificação de outras instâncias relevantes a aplicações reais.

1.4 Estrutura do documento

Os capítulos seguintes deste documento encontram-se organizados da seguinte forma:

- *Capítulo 2 - Consultas por similaridade*: explana a teoria de espaços métricos, funções de distância e operadores de seleção e junção por similaridade, além de descrever os trabalhos correlatos presentes na literatura.
- *Capítulo 3 - QuickNearest*: introduz o algoritmo *QuickNearest*, uma técnica eficiente para processamento de junções por similaridade do tipo kNN , e que visa auxiliar a resolver o primeiro objetivo desta tese.
- *Capítulo 4 - Wide-join*: descreve o operador *wide-join*, que visa auxiliar a resolver o segundo objetivo desta tese. O *wide-join* é um operador de junção com predicado parametrizável e que permite associar os conceitos de junção e ordenação de dados a fim de produzir resultados mais intuitivos e de melhor qualidade que os operadores de junção tradicionais.
- *Capítulo 5 - Conclusão*: apresenta as conclusões do trabalho e delineia algumas metas de desenvolvimento futuros que podem dar continuidade a este trabalho.

Consultas por similaridade

Com o crescimento dos dados armazenados em meios digitais, tanto em volume quanto em variedade, repositórios cada vez maiores são necessários para seu armazenamento. Para que os usuários sejam capazes de acessar um determinado dado, os dados precisam ser estruturados e manipulados de maneira eficiente e eficaz [93]. Contudo, principalmente com relação a dados multimídia, os repositórios de dados não permitem que estruturas rígidas sejam criadas, como se fazia quando os dados eram apenas os escalares, pois novos tipos de dados tais como textos longos, imagens, áudio e vídeo podem ser armazenados e buscados, não sendo possível organizar a informação em chaves e registros [24], como nos SGBDRs tradicionais. Tão grande é a variedade entre esses objetos que eles não podem nem mesmo ser comparados pelos operadores relacionais usuais ($=$, $\neq$, $<$ etc.) [33]. Dessa forma, o principal critério de comparação entre essas informações é uma medida do quão parecido um objeto é do outro. À busca por objetos seguindo esse critério de comparação dá-se o nome de *consulta por similaridade*, que é explorada neste capítulo.

A seção 2.1 introduz os conceitos básicos sobre similaridade. A seção 2.2 descreve os operadores e condições de comparação por similaridade. A seguir, os operadores de seleção (seção 2.3) e junção (seção 2.4) por similaridade são abordados e formalizados. A seção 2.5 discute os estudos atuais envolvendo os operadores de junção por similaridade. Na seção 2.6 são apresentados os operadores físicos de junção. As considerações finais do capítulo são expostas na seção 2.7.

2.1 Modelo de similaridade

Consultas por similaridade podem ser vistas como o processo de recuperação da informação onde a busca é expressa por um *objeto complexo* e a resposta consiste em um conjunto de elementos considerados, de algum modo, similares ao objeto de consulta [44], bem como de informações associadas a cada elemento retornado. Dessa forma, para que seja possível recuperar elementos em resposta a uma consulta, o relacionamento entre esses elementos e o centro de consulta é estabelecido através de uma medida de similaridade [11].

Para computar a medida de similaridade, um elemento complexo é usualmente representado por um conjunto de valores que descrevem suas características, chamado *vetor de características*. Os algoritmos que processam um dado complexo para obter valores numéricos representativos de suas propriedades são chamados de *extratores de características*. Para cada domínio distinto de dados complexos existem características particulares que são consideradas na representação dos dados, e, por consequência, extratores de características específicos. Por exemplo, no domínio de imagens, as características mais comumente extraídas são o histograma de cor e valores representativos de forma ou de textura [29]. Já no domínio de áudio, a extração de características é baseada na variação temporal da intensidade de frequências do sinal de áudio, sendo que os extratores costumam empregar transformadas, como a Transformada de Fourier de Curta Duração ou a Transformada de Wavelet [6].

Independente do tipo de extrator utilizado, um dado complexo é geralmente representado por um vetor de características na forma $[v_1, v_2, \dots]$, em que cada valor v_i descreve uma característica inerente ao dado. Por sua vez, a avaliação da medida de similaridade entre vetores de características é realizada usando uma função que quantifica a diferença entre dois vetores quaisquer em um número real não-negativo. A essa função matemática, expressa pela definição 1 [30], dá-se o nome de *função de distância*.

Definição 1 (Função de distância). *Seja $\mathbb{D}$ um domínio de dados. Uma função $d : \mathbb{D} \times \mathbb{D} \mapsto \mathbb{R}_+$ é chamada função de distância se, $\forall x, y \in \mathbb{D}$, ela satisfaz às seguintes propriedades:*

- $d(x, y) \geq 0$ (não-negatividade);
- $d(x, y) = d(y, x)$ (simetria);
- $d(x, x) = 0$ (reflexividade);

É interessante observar que a literatura de Bases de Dados e de Consultas por Similaridade não traz a definição formal do que é um dado complexo, mas apenas apresenta exemplos de dados complexos. Diversos estudos apenas citam que “*bancos de dados são bem equipados para recuperar números, datas e strings, porém, é necessário que passem a manipular outros tipos de dados, tais como imagens, vídeos, áudio etc, genericamente chamados de dados complexos*” [35, 7, 50, 17, 70]. No contexto desta tese, tendo com base a definição de função de distância, o termo dado complexo é formalmente expresso pela definição 2.

Definição 2 (Dado complexo). *Dados complexos são aqueles que são comparados usando uma função de distância.*

Embora a definição de função de distância seja suficiente para estabelecer a medida de similaridade entre dois elementos distintos, consultas por similaridade em geral empregam uma função de distância especial que, além das três propriedades elencadas, respeita uma quarta propriedade conhecida por *desigualdade triangular*. A essa função de distância especial dá-se o nome de *métrica*.

Formalmente, uma métrica é expressa de acordo com a definição 3 [57]. O par ordenado $\mathcal{M} = \langle \mathbb{D}, d \rangle$, onde $\mathbb{D}$ é um domínio de dados e d é uma métrica, é chamado de *espaço métrico*.

Definição 3 (Métrica). *Seja $\mathbb{D}$ um domínio de dados. Uma função de distância $d : \mathbb{D} \times \mathbb{D} \mapsto \mathbb{R}_+$ é chamada métrica se, $\forall x, y, z \in \mathbb{D}$, ela satisfaz às seguintes propriedades:*

- $d(x, y) \geq 0$ (não-negatividade);
- $d(x, y) = 0 \Leftrightarrow x = y$ (identidade dos indiscerníveis);
- $d(x, y) = d(y, x)$ (simetria);
- $d(x, y) \leq d(x, z) + d(z, y)$ (desigualdade triangular).

A literatura é rica em funções de distâncias [30] que podem ser utilizadas para computar a similaridade entre dois elementos. Para aplicações em consultas por similaridade sobre dados representados em espaços multidimensionais, uma das mais utilizadas é a função conhecida como Minkowski.

A métrica Minkowski forma uma coleção de funções matemáticas definidas para espaços normados, denotadas pelas respectivas normas L_p , sendo que cada valor distinto do parâmetro p designa uma métrica individual [93]. Uma função L_p é definida de acordo com a equação 2.1, onde $p \geq 1$ é um valor inteiro e n é a dimensão do vetor de características de dois elementos complexos x e y do espaço. Para ser possível aplicar as distâncias da família Minkowski a dois elementos, ambos os vetores de características devem ter a mesma dimensionalidade, isto é, a mesma quantidade de características.

$$L_p([x_1, \dots, x_n], [y_1, \dots, y_n]) = \sqrt[p]{\sum_{i=1}^n |x_i - y_i|^p} \quad (2.1)$$

Para ilustrar, e sem perda de generalidade, a figura 2.1 assume um espaço bidimensional onde se mostra o formato da região delimitada pela equação 2.1 para diversos valores do parâmetro p . Alguns valores de p correspondem à instâncias especiais da métrica Minkowski. Por exemplo, $p = 1$ define a *métrica Manhattan*, também chamada de *City Block* que é representada na figura 2.1(b). A bem conhecida *distância Euclidiana*, ilustrada na figura 2.1(c), é obtida quando $p = 2$. Já a distância L_∞ (figura 2.1(e)) é chamada de *distância de Chebyshev* ou *métrica máxima*. A figura 2.1(d) mostra ainda a métrica L_8 , obtida com o valor de $p = 8$ na equação 2.1. Note-se que para valores de p menores que 1, a equação 2.1 não gera métrica, pois a função resultante não respeita a desigualdade triangular. A figura 2.1(a) mostra o formato da região gerada quando $p = 0, 5$.

Outro exemplo de métrica, frequentemente utilizada para espaços de sequências de símbolos, é a chamada *distância de edição* [52]. Esta métrica compara sequências (como palavras, sequências genéticas etc.) e pode operar sobre objetos em espaços adimensionais. A distância de edição calcula o número mínimo de operações atômicas de edição (inserção, remoção ou substituição de um símbolo) necessárias para transformar uma sequência em outra. Essas operações são expressas como segue [93]:

- *inserção*(x, i, c): insere o símbolo c na posição i da sequência x .

Figura 2.1: Formato da região delimitada pela equação 2.1 para diversos valores do parâmetro p em um espaço bidimensional

- *remoção*(x, i): exclui o símbolo da posição i da sequência x .
- *substituição*(x, i, c): substitui o símbolo na posição i da sequência x pelo novo símbolo c .

Por exemplo, a distância de edição entre as palavras ‘flora’ e ‘flor’ é 1, que corresponde à remoção da letra ‘a’.

Existem também métricas capazes de computar a similaridade entre dois conjuntos de elementos em espaços de conjuntos. O exemplo mais comum é a métrica conhecida como *distância de Jaccard*. O valor de similaridade computado por esta distância é baseado na razão entre as cardinalidades da intersecção e da união dos dois conjuntos envolvidos [93]. Assumindo dois conjuntos de elementos X e Y , a distância de Jaccard é expressa de acordo com a equação 2.2.

$$d(X, Y) = 1 - \frac{|X \cap Y|}{|X \cup Y|} \quad (2.2)$$

Uma vez que dados complexos normalmente não estão sujeitos às propriedades das RI ou RO, o modelo de similaridade definido através de espaços métricos torna-se adequado para representar as relações de similaridade (RS). Eles podem ser comparados pela medida de similaridade estabelecida pela métrica empregada, sendo que um valor de similaridade bem próximo a zero indica elementos muito similares, enquanto que um valor exatamente igual a zero significa que ambos os elementos são totalmente idênticos. As medidas de similaridade retornadas pelas métricas são utilizadas pelos *operadores de comparação por similaridade*, discutidos na seção 2.2 a seguir.

2.2 Operadores de comparação por similaridade

Um *operador relacional de consulta* recebe como entrada uma ou mais relações e retorna como saída uma relação. Esses operadores são utilizados para recuperar dados nos SGBDRs. A álgebra relacional define seis *operadores fundamentais*: união ($\cup$), diferença ($-$), produto Cartesiano ($\times$), projeção (π), seleção (σ) e um operador para renomear atributos e relações (ρ). Todos os demais operadores, ditos *operadores derivados*, podem ser obtidos a partir de combinações destes seis operadores fundamentais. Por exemplo, uma junção é definida como sendo o produto Cartesiano seguido de uma seleção, enquanto o operador de intersecção entre dois conjuntos equivale à diferença do primeiro conjunto com a diferença do primeiro pelo segundo: $A \cap B \Leftrightarrow A - (A - B)$ [32].

Todas as consultas são expressas pela combinação dos operadores de busca, tanto os fundamentais quanto os derivados. Os operadores derivados, além de facilitar a formulação das consultas, trazem vantagens de implementação, pois os algoritmos que os processam tendem a ser mais rápidos que aqueles obtidos pela composição de operadores fundamentais [28].

Além dos operadores fundamentais e derivados existem outros operadores, ditos *operadores estendidos*, que são necessários para implementar SGBDs baseados no modelo relacional. Estes operadores não são relacionais, entretanto, de modo análogo ao operador renomear (ρ), eles são úteis para ferramentas práticas envolvendo situações reais. Como exemplos de operadores estendidos, tem-se os operadores de extensão e de ordenação [39, 28, 42], apresentados a seguir:

- Operador de extensão ($\psi_{\{e/A\}}R$): adiciona o atributo A à relação R e em cada tupla r atribui o valor do atributo $r.A$ como sendo o resultado da expressão e , que é calculada com base nos valores dos demais atributos da tupla.
- Operador de ordenação ($\omega_{\{L\}}R$): ordena as tuplas da relação R pelos valores da lista de atributos L . É importante notar que a ordenação é um operador de extensão, fora da álgebra relacional, e qualquer operador executado depois dele numa expressão de consulta não garante que a ordem seja preservada.

Dentre os operadores da álgebra relacional que permitem realizar a recuperação de tuplas de relações, os operadores de seleção, junção e agrupamento (γ) são aqueles que fazem comparações de valores de atributos nas tuplas. As operações de comparação de elementos em SGBDRs são baseadas em *predicados de comparação*. Formalmente, um predicado é expresso de acordo com a definição 4. Os predicados fazem uso de operadores de comparação para determinar como as tuplas resultantes de uma consulta são selecionadas [63].

Definição 4 (Predicado). *Um predicado é uma expressão em lógica de 3 valores na forma $p = \varphi\tau_1\varphi \dots \varphi\tau_n$, onde φ é um operador de conjunção ($\wedge$), disjunção ($\vee$) ou negação ($\neg$), e τ_i é um termo na forma $A \theta v$, onde A é um atributo da relação envolvida, v é o valor com o qual o atributo A é comparado (v é amostrado no domínio de A) e θ é um operador de comparação definido sobre o domínio de A .*

A avaliação de um predicado p resulta no valor TRUE (verdadeiro) sempre que a condição expressa por $(A \theta v)$ for satisfeita, e o valor FALSE (falso) caso contrário. Entretanto, SGBDRs podem possuir tuplas em que determinados atributos não contenham valor, ou seja, estejam em estado NULL (nulo). Quando um dos valores comparados é NULL, o resultado da avaliação do predicado p é UNKNOWN (desconhecido) [28]. Por exemplo, a expressão $(\text{NULL} > 4)$ tem UNKNOWN como resposta.

Uma vez que é comum que um predicado seja formado por uma sequência de termos, devido à presença do valor UNKNOWN entre as possibilidades de resposta de cada termo τ , a avaliação dos operadores de conjunção e disjunção é dada de acordo com as tabelas 2.1 e 2.2, respectivamente. Já a tabela 2.3 mostra o emprego do operador de negação sobre um termo em lógica de 3 valores [28].

Em SGBDRs, as comparações mais frequentes são baseadas em Relação de Igualdade (RI), onde θ corresponde aos operadores $=$ ou $\neq$. Entretanto, comparações por relações de ordem entre os elementos também são úteis. Para isso, operadores como $\leq$ e $\geq$ fazem uso da propriedade

Tabela 2.1: Conjunção

τ_1	τ_2	$\tau_1 \wedge \tau_2$
TRUE	TRUE	TRUE
TRUE	UNKNOWN	UNKNOWN
TRUE	FALSE	FALSE
UNKNOWN	TRUE	UNKNOWN
UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	FALSE	FALSE
FALSE	TRUE	FALSE
FALSE	UNKNOWN	FALSE
FALSE	FALSE	FALSE

Tabela 2.2: Disjunção

τ_1	τ_2	$\tau_1 \vee \tau_2$
TRUE	TRUE	TRUE
TRUE	UNKNOWN	TRUE
TRUE	FALSE	TRUE
UNKNOWN	TRUE	TRUE
UNKNOWN	UNKNOWN	UNKNOWN
UNKNOWN	FALSE	UNKNOWN
FALSE	TRUE	TRUE
FALSE	UNKNOWN	UNKNOWN
FALSE	FALSE	FALSE

Tabela 2.3: Negação

τ	$\neg \tau$
TRUE	FALSE
UNKNOWN	UNKNOWN
FALSE	TRUE

conhecida como Relação de Ordem Linear (ROL - definição 5) [12]. Para cada operador que obedeça à ROL, sempre existirá um operador dual atendendo às propriedades da Relação de Ordem Linear Estrita (ROLE - definição 6) [12], sendo que aqueles comumente suportados pelos SGBDRs são os operadores $<$ e $>$.

Definição 5 (Ordem Linear). *Dado um par ordenado $\langle \mathbb{X}, \leq \rangle$, onde $\mathbb{X}$ é um conjunto de elementos e $\leq$ é uma relação sobre $\mathbb{X}$, o conjunto $\mathbb{X}$ é dito estar em ordem linear se, $\forall a, b, c \in \mathbb{X}$, as seguintes propriedades se mantêm:*

- $a \leq a$ (reflexividade);
- $a \leq b \wedge b \leq a \Rightarrow a = b$ (antissimetria);
- $a \leq b \wedge b \leq c \Rightarrow a \leq c$ (transitividade);
- $a \leq b \vee b \leq a$ (comparabilidade).

Definição 6 (Ordem Linear Estrita). *Dado um par ordenado $\langle \mathbb{X}, < \rangle$, onde $\mathbb{X}$ é um conjunto de elementos e $<$ é uma relação sobre $\mathbb{X}$, o conjunto $\mathbb{X}$ é dito estar em ordem linear estrita se, $\forall a, b, c \in \mathbb{X}$, as seguintes propriedades sempre forem válidas:*

- $\neg(a < a)$ (irreflexividade);
- $a < b \Rightarrow \neg(b < a)$ (assimetria);
- $a < b \wedge b < c \Rightarrow a < c$ (transitividade);
- $a < b \vee b < a$ (comparabilidade).

Quando um operador de comparação $\{<, \leq, \dots\}$ é aplicado sobre um domínio numérico, como $\mathbb{I}$, $\mathbb{Q}$ ou mesmo $\mathbb{R}$, usa-se os símbolos $\{<, \leq, \dots\}$ respectivamente, como é o caso dos SGBDs e das linguagens de consulta. Por extensão, esses símbolos também são usados para comparar sequências de caracteres usando a ordem lexicográfica, instantes de tempo usando a precedência temporal entre dois eventos, e períodos de tempo, usando a duração de cada período. Dessa maneira, os operadores de comparação por identidade $\theta_i \in \{=, \neq\}$ e por ordem $\theta_o \in \{<, \leq, >, \geq\}$ formam os

operadores *big six* da linguagem SQL, e todos eles são aplicáveis a qualquer atributo A de qualquer domínio escalar atendido pelos SGBDR tradicionais. Além dos operadores *big six*, existem outros que podem ser aplicados a domínios específicos. Por exemplo, podem ser usados operadores de continência de subsequências ($\subset$, $\supset$, $\subseteq$ e $\supseteq$), quando o domínio é de cadeia de caracteres ou de períodos de tempo. Além disso, extensões dos SGBDR feitas para atender a domínios de dados específicos também acrescentam outros operadores de comparação. Por exemplo, extensões para tratamento de dados georreferenciados acrescentam operadores para comparar coordenadas geográficas, poligonais e áreas [60, 3].

Os predicados que devem comparar dados complexos em geral baseiam-se em critérios de *comparação por similaridade*. Nesta tese, para distinguir entre os atributos que amostram dados de domínios escalares, tradicionais, e os atributos que amostram dados de domínios complexos, os símbolos E_i são empregados para indicar atributos de domínios escalares $\mathbb{E}_i$ e S_i para indicar atributos de domínios complexos $\mathbb{S}_i$. Os símbolos A_i e $\mathbb{A}$ continuam a ser usados para indicar atributos e domínios quaisquer, quando não é importante distinguir se o atributo é escalar ou complexo. Assim, uma relação R é vista como um conjunto de tuplas cujo esquema é denotado por $R = (E_1, \dots, E_m, S_1, \dots, S_n)$. Os atributos escalares E_i são normalmente comparados pelos operadores *big six* (baseados em relações de identidade e de ordem), enquanto os atributos complexos S_i são comparados por operadores baseados em relações de identidade ou de similaridade.

Os operadores de comparação por similaridade empregam uma função de distância $d : \mathbb{S} \times \mathbb{S} \mapsto \mathbb{R}_+$ para comparar cada par de elementos complexo de um mesmo domínio, isto é, $s_1, s_2 \in \mathbb{S}$. Tal como qualquer operador de comparação, os operadores por similaridade são expressos em termos da forma $\tau_s = (s_1 \theta_s s_2)$. Assim, a função de distância usada é indicada como parte do operador θ_s , junto com os demais parâmetros usados na comparação. Existem dois operadores de comparação por similaridade tratados com mais frequência na literatura [14, 82, 7]: *similarity range* e *k-nearest neighbor*. Eles são denotados como:

- *Similarity range*: $\theta_s = \text{Rng}(d, \xi)$ – este operador retorna TRUE para a comparação $(s_1 \text{Rng}(d, \xi) s_2)$, $\forall s_i \in \mathbb{S}$, sempre que $d(s_1, s_2) \leq \xi$, onde $\mathbb{S}$ é o espaço de busca, ou seja, o conjunto que deve ser pesquisado..
- *k-nearest neighbor*: $\theta_s = \text{kNN}(d, k)$ – este operador retorna TRUE para a comparação $(s_1 \text{kNN}(d, k) s_2)$, $\forall s_i \in \mathbb{S}$, sempre que $d(s_1, s_2)$ for uma das k menores distâncias $d(s_1, s_i)$, onde $\mathbb{S}$ é o espaço de busca, ou seja, o conjunto que deve ser pesquisado.

O operador de comparação *similarity range* é um operador *orientado à tupla*, isto é, dados dois elementos $s_i, s_q \in \mathbb{S}$, um predicado $(s_i \text{Rng}(d, \xi) s_q)$ pode ser avaliado sem qualquer informação adicional sobre os outros elementos contidos na base de dados. Por sua vez, o operador *k-nearest neighbor* é dito ser *orientado ao espaço de busca*, uma vez que ele necessita conhecer todos os elementos presentes no espaço de busca para poder arbitrar sobre os k mais similares em um predicado $(s_i \text{kNN}(d, k) s_q)$.

Entre os operadores da álgebra relacional, os operadores de seleção e junção são os mais flexíveis, permitindo comparar atributos usando predicados baseados tanto em RI, ROL ou ROLE, bem

como por RS em algumas extensões para incorporar o conceito de similaridade sobre o modelo relacional, tais como os protótipos *SimDB - Similarity Aware Database* [72], *Mufin - Multi-feature indexing* [92], *MSQL - Metric SQL* [59], *PostgresIE - Postgres Image Extension* [43], *SiREn - Similarity Retrieval Engine* [8, 7], *FMI-SiR - Features, Metrics and Indexes for Similarity Retrieval* [50], *SimBA - Similarity by Authority* [9] e *Kiara* [66]. Por sua vez, o operador de agrupamento também avalia predicados, mas apenas predicados baseados em RI podem ser usados [41, 73, 95]. Quando o predicado empregado nos operadores de seleção e junção faz uso dos operadores de comparação por similaridade, eles correspondem aos operadores de seleção por similaridade e aos operadores de junção por similaridade, descritos, respectivamente, nas seções 2.3 e 2.4.

2.3 Seleção por similaridade

O operador de seleção é usualmente representado por $\sigma_{(p)}R$. Este operador é aplicado a uma única relação R e produz uma nova relação, que é formada por um subconjunto das tuplas da relação R . As tuplas da relação resultante são aquelas que satisfazem o predicado p sobre as tuplas de R . O emprego de cada um dos dois operadores de comparação por similaridade ao predicado p gera duas instâncias de operadores de seleção por similaridade: seleção por abrangência (*range selection*) e seleção pelos k vizinhos mais próximos (*k-nearest neighbor selection*).

O operador *range selection* recupera todos os elementos da base de dados que sejam dissimilares ao objeto de consulta a até no máximo um certo limiar, sendo a medida de similaridade calculada por uma função de distância. Um *range selection* pode ser formalmente expresso de acordo com a definição 7.

Definição 7 (*Range selection*). *Seja $\mathcal{S}$ um domínio de dados complexos, R uma relação, $S \subseteq R$ um atributo complexo amostrado no domínio $\mathcal{S}$, d uma função de distância e $\xi \in \mathbb{R}_+$ um limiar de similaridade. O operador de seleção por similaridade *range selection* sobre um centro de consulta $s_q \in \mathcal{S}$ é dado por:*

$$\sigma_{(S \text{Rng}(d, \xi) s_q)} R = \{r \in R \mid d(s_q, r.S) \leq \xi\}$$

O pseudocódigo de um algoritmo para executar *range selection* em um conjunto finito de elementos complexos é apresentado no algoritmo 2.1. Embora um *range selection* seja o tipo de consulta por similaridade mais básica [24], este operador necessita que seja especificado um valor ξ significativo para a consulta. Porém, esse valor não estabelece um limite superior para a cardinalidade do conjunto-resposta. Além disso, nem sempre é fácil ou intuitivo determinar o valor de ξ , sendo muitas vezes necessário conhecimento do domínio. Caso não se tenha noções sobre a distribuição dos dados, um *range selection* pode resultar em um conjunto-resposta com cardinalidade muito pequena (até nula) ou muito grande [14].

Visando contornar o problema de escolher um valor adequado para o parâmetro ξ , uma segunda seleção por similaridade (*k-nearest neighbor selection*) permite especificar diretamente o número de elementos incluídos no conjunto resposta [10].

Algoritmo 2.1: *Range selection*

Entrada: A relação R ; o limiar de similaridade ξ ; o centro de consulta $s_q \in \mathbb{S}$; a função de distância d .

Saída : O conjunto-resposta T com os elementos de R com distância máxima ξ a s_q .

```

1  $T \leftarrow \emptyset$ ;
2 para  $r \in R$  faça
3   se  $d(s_q, r.\mathcal{S}) \leq \xi$  então
4   |    $T \leftarrow T \cup \{r\}$ ;
5   fim
6 fim
7 retorna  $T$ ;
```

Um *k-nearest neighbor selection* recupera uma quantidade k de elementos armazenados na base de dados que sejam os mais similares ao objeto de consulta, sendo a similaridade computada por uma função de distância. Por definição, assume-se que o espaço de busca para o operador de comparação *k-nearest neighbor* usado no operador de busca *k-nearest neighbor selection* é o domínio ativo do respectivo conjunto de atributos na relação. Com isso, um *k-nearest neighbor selection* é expresso de acordo com a definição 8 [10].

Definição 8 (*k-nearest neighbor selection*). *Seja $\mathbb{S}$ um domínio de dados complexos, R uma relação, $\mathcal{S} \subseteq R$ um atributo complexo amostrado no domínio $\mathbb{S}$, d uma função de distância e $k \in \mathbb{N}^*$ uma constante. O operador de seleção por similaridade *k-nearest neighbor selection* sobre um centro de consulta $s_q \in \mathbb{S}$ é dado por:*

$$\sigma_{(S^{kNN}(d,k)_{s_q})} R = \begin{cases} NNS_0 : \emptyset \\ NNS_k : \{r \in R \mid \forall t \in (R \setminus NNS_{(k-1)}) : d(s_q, r.\mathcal{S}) \leq d(s_q, t.\mathcal{S})\} \end{cases}$$

Analisando a relação de recorrência da definição 8, pode-se perceber que um *k-nearest neighbor selection* é obtido de modo incremental. Entretanto, pode acontecer de mais de um elemento ocupar a k -ésima posição da busca. Neste caso, há duas opções: a primeira é considerar todos os elementos empatados como pertencendo ao conjunto resposta ou, então, escolher apenas um deles, de maneira aleatória, para ocupar a k -ésima posição, sendo que esta segunda solução foi empregada nas implementações resultantes desta tese. O algoritmo 2.2 permite executar este tipo de seleção em um conjunto finito de elementos complexos.

As operações de seleção do tipo *range* e *k-nearest neighbor* são ilustradas, respectivamente, pelas figuras 2.2(a) e 2.2(b).

Os operadores de seleção por similaridade atuam sobre uma única relação; entretanto, os operadores de junção, descritos na próxima seção 2.4, processam duas relações numa mesma operação.

Figura 2.2: Operadores de seleção por similaridade: as estrelas representam os centros de consulta e os losangos são os elementos selecionados na busca.

2.4 Junção por similaridade

O operador de junção também é muito útil no processamento de consultas e vem sendo extensivamente estudado. Formalmente, o operador de junção pode ser expresso de acordo com a definição 9 [39, 71, 32].

Definição 9 (Junção). *A junção entre duas relações R e S é uma operação relacional binária, algebricamente definida como a composição de duas operações fundamentais: um produto Cartesiano seguido por uma seleção, que define o predicado de junção, da seguinte maneira:*

$$R \bowtie^{(p)} S \Leftrightarrow \sigma_{(p)}(R \times S)$$

Algoritmo 2.2: k -nearest neighbor selection

Entrada: A relação R ; uma constante inteira positiva k ; o centro de consulta $s_q \in \mathbb{S}$; a função de distância d .

Saída : Um conjunto-resposta T com os k elementos de R mais similares a s_q .

```

1  $T \leftarrow \emptyset$ ;
2 para  $r \in R$  faça
3   se  $|T| < k$  então
4      $T \leftarrow T \cup \{r\}$ ;
5   senão
6      $x \leftarrow \max(T)$ ;                                /*  $x \in T$  é o elemento mais dissimilar a  $s_q$  */
7     se  $d(s_q, r.S) \leq d(s_q, x)$  então
8        $T \leftarrow T \cup \{r\}$ ;
9        $T \leftarrow T \setminus \{x\}$ ;
10    fim
11  fim
12 fim
13 retorna  $T$ ;

```

Alguns trabalhos, como os apresentados em [64, 41], são voltados para a execução de junção sobre dados escalares. Com relação ao processamento de junções por similaridade, a literatura frequentemente assume que existem três variantes desse operador [14, 75, 76]: *range join*, *k-nearest neighbor join* e *k-closest neighbor join*.

O operador *range join* (figura 2.3(a)) utiliza o operador de comparação *similarity range* para compor seu predicado. Dessa forma, cada elemento complexo pertencente à primeira relação de junção é combinado com os elementos da segunda relação, e os pares que constituem a resposta são aqueles em que a dissimilaridade entre os elementos comparados não seja maior que o limiar ξ . Este operador pode ser aplicado, por exemplo, para problemas de casamento de cadeias de caracteres (*string matching*) [68, 83], detecção de quase-duplicatas (*near duplicate detection*) [84, 85, 19] e *data cleaning* [23]. Analogamente a um *range selection*, a determinação do parâmetro ξ nem sempre é intuitiva. Além disso, como o mesmo limiar ξ é usado sobre todo o espaço, variações de densidade da distribuição de dados dificultam ainda mais a obtenção de respostas adequadas. O operador *range join* é expresso pela definição 10.

Definição 10 (*Range join*). *Seja $\mathcal{S}$ um domínio de dados complexos, R e S duas relações, $\mathcal{S}_1 \subseteq R$ e $\mathcal{S}_2 \subseteq S$ dois atributos complexos no domínio $\mathcal{S}$, $r \in R$ e $s \in S$ tuplas contendo atributos complexos, d uma função de distância e $\xi \in \mathbb{R}_+$ um limiar de similaridade. O operador *range join* é expresso por:*

$$R \stackrel{(\mathcal{S}_1 \text{ Rng}(d, \xi) \mathcal{S}_2)}{\bowtie} S = \{\langle r, s \rangle \in (R \times S) \mid d(r.\mathcal{S}_1, s.\mathcal{S}_2) \leq \xi\}$$

As outras duas variantes do operador de junção por similaridade baseiam-se numa contagem fixa de elementos a serem recuperados. Da mesma maneira que para o operador de busca *k-nearest neighbor selection*, assume-se por definição para esses dois operadores de junção por similaridade que o espaço de busca para o operador de comparação *k-nearest neighbor* é o domínio ativo do respectivo atributo na relação da direita.

O operador *k-nearest neighbor join* (figura 2.3(b)) compara cada elemento da primeira relação com os k elementos da segunda, sendo que a resposta inclui os elementos complexos da segunda relação que são os k mais similares a cada elemento da primeira. O conjunto-resposta retornado por este operador tem cardinalidade igual a cardinalidade da primeira relação multiplicada por k , ou seja, o operador recupera $(k \cdot |R|)$ tuplas. Formalmente, um *k-nearest neighbor join* é expresso pela definição 11 [14, 35, 34].

Definição 11 (*k-nearest neighbor join*). *Seja $\mathcal{S}$ um domínio de dados complexos, R e S duas relações, $\mathcal{S}_1$ e $\mathcal{S}_2$ dois atributos complexos no domínio $\mathcal{S}$, $r \in R$ e $s, s' \in S$ tuplas, d uma função de distância e $k \in \mathbb{N}^*$ uma constante. O operador *k-nearest neighbor join* é convencionalmente expresso por:*

$$R \stackrel{(\mathcal{S}_1 \text{ kNN}(d, k) \mathcal{S}_2)}{\bowtie} S = T = \{\langle r, s \rangle \in (R \times S) \mid \forall \langle r, s' \rangle \in ((R \times S) \setminus T) \Rightarrow |T| = k \cdot |R| \wedge d(r.\mathcal{S}_1, s.\mathcal{S}_2) \leq d(r.\mathcal{S}_1, s'.\mathcal{S}_2)\}$$

O operador *k-closest neighbor join* (figura 2.3(d)) recupera uma quantidade k de pares de elementos de modo que a medida de similaridade entre os pares de elementos complexos tenham os menores valores. Este operador difere do operador *k-nearest neighbor join* por recuperar os k pares $\{\langle r, s \rangle, r \in R, s \in S\}$ mais similares entre si considerando *todo o conjunto em um contexto geral*, enquanto que um *k-nearest neighbor join* combina k elementos da segunda relação com *cada elemento* da primeira. Dessa forma, a cardinalidade do conjunto resposta de um *k-closest neighbor join* é igual ao valor k . Um *k-closest neighbor join* é expresso de acordo com a definição 12 [14].

Definição 12 (*k-closest neighbor join*). *Seja $\mathcal{S}$ um domínio de dados complexos, R e S duas relações, $\mathcal{S}_1$ e $\mathcal{S}_2$ dois atributos complexos no domínio $\mathcal{S}$, $r, r' \in R$ e $s, s' \in S$ tuplas, d uma função de distância e $k \in \mathbb{N}^*$ uma constante. O operador *k-closest neighbor join* é expresso por:*

$$R \stackrel{(\mathcal{S}_1 \text{ kCN}(d,k) \mathcal{S}_2)}{\bowtie} S = T = \{\langle r, s \rangle \in (R \times S) \mid \forall \langle r', s' \rangle \in ((R \times S) \setminus T) \Rightarrow |T| = k \wedge d(r.\mathcal{S}_1, s.\mathcal{S}_2) \leq d(r'.\mathcal{S}_1, s'.\mathcal{S}_2)\}$$

Ambos os operadores *k-nearest neighbor join* e *k-closest neighbor join* podem ser aplicados em operações de mineração e integração de dados [14], aplicações de otimização envolvendo *MapReduce* [94] e consultas sobre dados de alta dimensionalidade [88].

Existe, ainda, um quarto tipo de junção por similaridade conhecido por *join around* (figura 2.3(c)) [74]. O *join around* combina algumas das propriedades dos operadores *range* e *k-nearest neighbor joins* em um único operador, de modo que cada elemento complexo do primeiro conjunto é pareado com seu elemento mais similar (1-NN) no segundo conjunto. Além disso, o valor de dissimilaridade entre os dois elementos não pode ultrapassar o limiar ξ . De forma similar ao *range join*, não é possível estabelecer com exatidão a cardinalidade do conjunto resposta do operador *join around*. Entretanto, por este operador combinar os operadores de comparação *Rng*(d, ξ) e *kNN*($d, 1$) em um único predicado [75], essa cardinalidade é limitada superiormente pela cardinalidade do conjunto R . O operador *join around* pode ser formalizado de acordo com a definição 13.

Definição 13 (*Join around*). *O operador *join around* corresponde à combinação conjuntiva dos operadores *range join* e *k-nearest neighbor join*, com valor de $k = 1$. Formalmente:*

$$R \stackrel{(\mathcal{S}_1 \text{ Ard}(d,\xi) \mathcal{S}_2)}{\bowtie} S \Leftrightarrow \left(R \stackrel{(\mathcal{S}_1 \text{ Rng}(d,\xi) \mathcal{S}_2)}{\bowtie} S \right) \cap \left(R \stackrel{(\mathcal{S}_1 \text{ kNN}(d,1) \mathcal{S}_2)}{\bowtie} S \right)$$

2.5 Abordagens de junção por similaridade

Dos três operadores de junção por similaridade, o operador *range join* é o mais explorado pela literatura [31, 49, 74, 75, 77, 38, 65, 26, 25]. Os primeiros estudos cujo objeto é o operador *range join* [74] enfocavam a definição de regras algébricas e propriedades de equivalência entre os operadores de seleção, junção e agrupamento por similaridade, aplicando o operador de comparação *similarity*

Figura 2.3: Operadores de junção por similaridade: estrelas representam os elementos da primeira relação. Círculos representam os elementos da segunda. Losangos são os elementos da segunda relação selecionados na junção e pareados com os elementos da primeira relação.

range para compor os predicados das consultas. Estas ideias iniciais foram aperfeiçoadas em [75] e resultaram na proposta do operador *join around* (seção 2.4).

Uma extensão do trabalho apresentado em [75] é encontrada em [77]. Esse trabalho apresenta uma implementação física do operador *range join* baseada em laços aninhados e realiza sua integração ao módulo de processamento de consultas do banco de dados PostgreSQL.

Também baseado no operador *range join*, o trabalho apresentado em [61] aplica esse operador na definição de uma operação de intersecção por similaridade entre conjuntos. A operação de intersecção por similaridade considera diferentes limiares de similaridade $\xi_1, \dots, \xi_n$, um para cada atributo complexo das tuplas envolvidas na operação, e então aplica o predicado *similarity range* entre os elementos das duas relações, considerando um dos valores ξ_i para cada atributo complexo A_i presente nas relações. Por exemplo, sendo R e S duas relações com um único atributo complexo, onde os valores de atributos x e $\tilde{x}$ são considerados similares, isto é, $d(x, \tilde{x}) \leq \xi$, para $R = \{a, b, c, d, e, f, g, z\}$ e $S = \{\tilde{a}, \tilde{b}, \tilde{c}, h, i, j, k, l, z\}$, a intersecção por similaridade $R \cap S = \{a, b, c, \tilde{a}, \tilde{b}, \tilde{c}, z\}$ [61].

A principal limitação dos trabalhos mencionados está ligada à definição do parâmetro ξ . Como explanado em [14], a faixa de valores de ξ adequados para o processamento de um predicado do tipo *range* é relativamente estreita e quase sempre dependente de conhecimento prévio de especialistas no domínio ou da distribuição do conjunto de dados complexos. Se o valor do limiar de similaridade é escolhido muito baixo, o operador pode retornar um conjunto-resposta vazio,

enquanto que se o parâmetro ξ possuir um valor muito alto, esse limiar pode englobar todos os elementos presentes na base de dados, e todos os pares de elementos são retornados, de modo análogo a um produto Cartesiano.

Além disso, uma vez que ξ quase sempre não está associado a uma unidade de medida (por exemplo, a similaridade entre dois elementos de um espaço geográfico pode ser medida em k , mas não é intuitivo se referir à unidade de medida da similaridade entre dois histogramas), torna-se muito mais intuitivo executar consultas por similaridade que recuperem uma dada quantidade de elementos do que numa unidade de similaridade baseada apenas num limiar. Diferente dos estudos apresentados em [74, 75, 77, 61], que empregam o operador *range join*, a primeira contribuição desta tese tem seu foco voltado para o operador *kNN*, enquanto que a segunda introduz conceitos relativos ao controle da cardinalidade do resultado, mesmo para o operador do tipo *range*.

Os operadores *k-nearest neighbor join* e *k-closest neighbor join* foram primeiramente empregados para operações voltadas para mineração de dados [14], e também foram estudados para responder a consultas baseadas em dados multidimensionais. O trabalho apresentado em [3] investiga a combinação de dois predicados do tipo *kNN* para dados espaciais, para ambos os operadores de seleção e junção por similaridade. Já os estudos presentes em [89, 45] focam em processar, respectivamente, as variantes *nearest* e *closest* do operador de junção por similaridade através de técnicas incrementais. Os estudos mencionados são restritos a dados multidimensionais, enquanto que os algoritmos projetados nesta tese aplicáveis a dados multidimensionais e puramente métricos.

Uma maneira alternativa de processar seleções por similaridade do tipo *kNN* é, de forma adaptativa, computar um *range selection* cujo limiar de similaridade ξ seja cuidadosamente ajustado para retornar um conjunto-resposta com exatamente k elementos. Tal ajuste pode ser realizado pela estimativa do valor de ξ usando técnicas como a análise da dimensão fractal do conjunto [4, 78]. Para o operador *k-nearest neighbor join*, contudo, este método não pode ser aplicado. Uma vez que as distâncias entre cada elemento da primeira relação R para seu k -ésimo objeto correspondente na segunda relação S não são correlacionadas, não é possível estimar um limiar ξ que assegure a quantidade correta de pares no resultado final. Por exemplo, considerando um valor de $k = 3$ e o limiar ξ estimado na figura 2.3(a), nota-se que para um dos elementos do primeiro conjunto a consulta recuperou exatos 3 objetos do segundo conjunto, mas para o outro elemento, o mesmo valor de ξ não recupera a quantidade correta.

Todos os trabalhos mencionados anteriormente empregam a estratégia de laço aninhado para computar junções por similaridade, executando $|R| \cdot |S|$ comparações entre os elementos para produzir o resultado final. Entretanto, uma estrutura de índice pode ser utilizada como forma de melhorar esta técnica. A ideia básica da abordagem indexada é inserir os elementos de uma ou de ambas as relações de junção em uma estrutura de dados que acelere sua recuperação. O trabalho descrito em [31] introduz uma estrutura de dados métrica chamada *eD-Index* para computar junções por similaridade em espaços métricos. Esta estrutura é baseada em particionamento de dados por bolas, criadas de acordo com elementos de referência (pivôs), e, segundo seus autores, é capaz de reduzir pela metade o tempo de execução da junção. A maior restrição desta estrutura é ser útil para computar apenas *self range joins*, isto é, executa apenas a junção onde ambas as relações são a mesma.

Outra estrutura de dados proposta para acelerar o processamento do operador de junção por similaridade é a assim chamada *List of Twin Clusters* (LTC) [67]. Para cada relação de junção, a LTC computa uma lista de grupos de elementos que se sobrepõem no espaço, bem como uma matriz de distância dos elementos para os elementos centrais de cada grupo. Na sequência, o algoritmo de consulta da LTC faz uso da propriedade da desigualdade triangular para podar elementos durante uma busca. A LTC pode ser vista como mais abrangente que a estrutura *eD-Index* por computar operações *k-closest neighbor joins* além das *range joins*, já que a estrutura *eD-Index* é específica para o operador *range*. Entretanto, a desvantagem da LTC é que as duas relações de junção precisam ser indexadas conjuntamente, de modo que qualquer alteração em quaisquer das relações faz com que a estrutura de índice precise ser inteiramente reconstruída.

Embora as estruturas de índices mencionadas sejam as mais frequentemente encontradas na literatura, qualquer estrutura de dados capaz de manipular dados representados num espaço métrico pode ser empregada. Por exemplo, as propostas descritas em [45, 48] fazem uso da estrutura R-Tree, que pode ser utilizada para indexar dados complexos representados num espaço multidimensional. Já o estudo descrito em [58] utiliza uma estrutura de espalhamento (*hash*). Para buscas por similaridade, entretanto, técnicas baseadas em *hash* não são exatas, retornando apenas aproximações da resposta [93]. O trabalho apresentado em [38] introduz uma estrutura de índice para acelerar o processamento do operador *k-closest neighbor join*, mas a estrutura proposta é restrita a dados multidimensionais, não podendo ser aplicada a espaços métricos em geral. Por sua vez, a proposta descrita em [26] apresenta uma estrutura de índice para computar apenas *range joins*.

Nesta tese, avaliou-se a estrutura Slim-Tree [81] para indexar a relação interna de junção e acelerar o processamento do algoritmo de laço aninhado. Esta estrutura tem a vantagem de ser capaz de permitir computar todos os três tipos básicos de junção por similaridade, enquanto nenhum dos trabalhos anteriormente mencionados tem essa capacidade. Entretanto, nos experimentos descritos no capítulo 3, pode-se notar que a Slim-Tree não foi capaz de trazer melhorias significativas de desempenho, sendo superada pela primeira proposta desta tese, o algoritmo *QuickNearest*.

Como exemplos de abordagens que não fazem uso de indexação para computar junções por similaridade, destacam-se o *Epsilon Grid Order* (EGO) [13] e o algoritmo *Quickjoin* [49]. As duas técnicas realizam o particionamento e agrupamento dos elementos do espaço de busca através de uma abordagem baseada na estratégia de divisão e conquista. O algoritmo EGO cria um reticulado de tamanho fixo ϵ e aplica estratégias específicas para ordenar os elementos e atravessar o reticulado. Esta técnica, contudo, é específica para dados multidimensionais, não sendo aplicável a espaços métricos em geral.

A técnica de junção *Quickjoin* pode ser sumarizada pelas quatro rotinas apresentadas nos algoritmos 2.3 a 2.6 [49, 37]. Esta técnica permite otimizar o processamento do operador *range join* em dados multidimensionais através da divisão dos elementos do espaço de busca em grupos tão pequenos que possam ser eficientemente comparados pela estratégia de laço aninhado (método de força bruta). O método de particionamento (algoritmo 2.4) reparte o conjunto de dados em quatro subconjuntos parcialmente sobrepostos através da escolha aleatória de elementos de referência (pivôs - *pivots*). Estes conjuntos são ressubmetidos ao algoritmo 2.3 de forma recursiva. Entretanto, pode acontecer de objetos pertencentes a conjuntos opostos satisfazerem o predicado de junção,

Algoritmo 2.3: *quickjoin*

Entrada: A relação R ; o limiar de similaridade ξ ; a função de distância d .
Saída : O conjunto-resposta T .

- 1 **se** $|R| \leq c$ **então retorna** $\text{joinBruteForce}(R, R, \xi)$;
- 2 $p_1 \leftarrow \text{randomElement}(R)$;
- 3 $p_2 \leftarrow \text{randomElement}(R - \{p_1\})$;
- 4 $\rho \leftarrow d(p_1, p_2)$;
- 5 $(L, G, L^w, G^w) \leftarrow \text{partition}(R, p_1, \xi, \rho)$;
- 6 **retorna** $T \leftarrow \text{quickjoinWin}(L^w, G^w, \xi) \cup \text{quickjoin}(L, \xi) \cup \text{quickjoin}(G, \xi)$;

Algoritmo 2.4: *partition*

Entrada: A relação R ; o pivô p ; o limiar de similaridade ξ ; a distância ρ entre os pivôs; a função de distância d .
Saída : Quatro partições de elementos.

- 1 $L \leftarrow \{r \in R \mid d(r, p) < \rho\}$;
- 2 $G \leftarrow \{r \in R \mid d(r, p) \geq \rho\}$;
- 3 $L^w \leftarrow \{r \in L \mid \rho - \xi \leq d(r, p)\}$;
- 4 $G^w \leftarrow \{r \in G \mid d(r, p) \leq \rho + \xi\}$;
- 5 **retorna** (L, G, L^w, G^w) ;

Algoritmo 2.5: *quickjoinWin*

Entrada: A relação R ; a relação S ; o limiar de similaridade ξ ; a função de distância d .
Saída : O conjunto-resposta T .

- 1 **se** $|R| + |S| \leq c$ **então retorna** $\text{joinBruteForce}(R, S, \xi)$;
- 2 $p_1 \leftarrow \text{randomElement}(R \cup S)$;
- 3 $p_2 \leftarrow \text{randomElement}((R \cup S) - \{p_1\})$;
- 4 $\rho \leftarrow d(p_1, p_2)$;
- 5 $(L_1, G_1, L_1^w, G_1^w) \leftarrow \text{partition}(R, p_1, \xi, \rho)$;
- 6 $(L_2, G_2, L_2^w, G_2^w) \leftarrow \text{partition}(S, p_1, \xi, \rho)$;
- 7 **retorna** $T_1 \leftarrow \text{quickjoinWin}(L_1^w, G_2^w, \xi) \cup \text{quickjoinWin}(G_1^w, L_2^w, \xi) \cup \text{quickjoinWin}(L_1, L_2, \xi) \cup \text{quickjoinWin}(G_1, G_2, \xi)$;

devendo ser combinados. A fim de encontrar tais pares de objetos, as partições finais computadas pelo método de particionamento são novamente submetidas à operação de junção (algoritmo 2.5). Ao final do processamento recursivo, o algoritmo retorna a resposta exata.

A principal diferença do algoritmo *Quickjoin* para as duas contribuições desta tese é que a primeira é voltada para o operador *k-nearest neighbor join*, enquanto que a segunda tem suas bases derivadas do operador *k-closest neighbor join*. Uma extensão do algoritmo *Quickjoin* capaz de computar predicados *kNN* é encontrada em [36, 37]. Esta extensão, contudo, computa apenas respostas aproximadas, onde alguns pares relevantes não são encontrados pelo algoritmo e são substituídos por outros de menor similaridade. Diferentemente destes estudos, esta tese obtém a resposta exata (não-aproximada) dos operadores de junção por similaridade.

O operador de junção por similaridade tem sido amplamente aplicado ao problema de casamento de cadeias de caracteres (*string matching*). Neste contexto, o principal interesse é recuperar as *strings* presentes na base de dados que são similares porém não idênticas. Diversos estudos presentes na literatura, como aqueles descritos em [55, 68, 83, 91, 90, 69] fazem uso de técnicas de

Algoritmo 2.6: *joinBruteForce*

Entrada: A relação R ; a relação S ; o limiar de similaridade ξ ; a função de distância d .

Saída : O conjunto-resposta T .

```

1  $T \leftarrow \emptyset$ ;
2 para cada  $r \in R$  faça
3   para cada  $s \in S$  faça
4     se  $d(r, s) \leq \xi$  então  $T \leftarrow T \cup \langle r, s \rangle$ ;
5   fim
6 fim
7 retorna  $T$ ;
```

particionamento e de estruturas de dados tais como árvores de sufixo, *tries* e *hashing* para resolver o problema. A limitação de todas estas propostas é que elas não são aplicáveis a outros tipos de dados exceto *strings*, enquanto que grande parte dos dados complexos são representados por vetores multidimensionais [93]. Além disso, elas focam apenas no operador *range join*.

Outras abordagens para o processamento do operador de junção por similaridade [94, 47, 25] empregam a técnica de *MapReduce*. A principal argumentação desta linha de trabalhos é que a computação atual está na era de “*big data*” e que os algoritmos que executam em um único processador, numa única *thread*, não são adequados para tratar grandes volumes de dados. Entretanto, nenhum destes trabalhos esclarece ou mensura o quanto de volume de dados é necessário para justificar explorar técnicas alternativas como *MapReduce*. Além disso, também não é explicado se as teorias e técnicas atualmente existentes (e.g. como divisão e conquista) foram exaustivamente exploradas antes de partir para métodos distribuídos, principalmente porque essas técnicas também podem ser usadas e contribuir para o aumento do desempenho do processamento paralelo.

Por exemplo, explorando a técnica de *MapReduce* em um *cluster* com 10 máquinas, o trabalho descrito em [25], apresenta resultados de uma junção por similaridade sobre 10 milhões de objetos em um tempo muito próximo àquele alcançado pelos resultados obtidos nesta tese (capítulo 3), para a mesma cardinalidade. É importante destacar que ambos os resultados não são diretamente comparáveis, pois em [25] foi empregado o operador *range join*, enquanto que os experimentos reportados no capítulo 3 apresentam os resultados de um *k-nearest neighbor join*. Além disso, existem diferenças quanto ao *hardware* utilizado na avaliação de ambos os trabalhos, sendo que o *hardware* da presente tese possui configuração inferior ao do referido trabalho. Entretanto, ressalta-se que a versão rodando em um único *thread*/processador apresentada nesta tese executou uma operação que é computacionalmente bem mais cara do que um *range join* tendo obtido quase um empate de desempenho em relação à versão distribuída de um operador bem mais simples. Assim, é oportuno avaliar se alternativas como o *MapReduce* são realmente necessárias para computar junções por similaridade, ou se os algoritmos existentes poderiam ser melhor projetados/refinados, tal como o algoritmo *QuickNearest* proposto no capítulo 3.

Ligadas à questão da ordenação discutida para o operador *k-closest neighbor join*, tem sido estudadas as consultas baseadas em ordenação, comumente chamadas (*ranked-based queries* ou *top-k queries*), onde as tuplas são classificadas de acordo com um critério de ordenação determinado pelo usuário. Consultas por similaridade, segundo este enfoque, procuram ordenar a relação de entrada e fazer com que os operadores do plano de consulta mantenham a ordenação dos dados de entrada.

Trabalhos como os encontrados em [1, 2, 53, 54] descrevem como são tratadas as relações ordenadas sobre o modelo relacional e introduzem álgebras de consulta que contemplem a ordem.

Como proposta inicial, em [1, 2] tem-se a descrição de relações ordenadas sobre o modelo relacional e a proposta de uma álgebra para consultar as relações definidas. Esse trabalho difere dos anteriores por permitir a classificação das tuplas baseando-se nos valores de um atributo ou na combinação de mais de um atributo, enquanto as demais abordagens estabeleciam um critério de classificação a partir de variantes de modelos matemáticos.

As propostas de [53, 54] estenderam as propriedades do modelo relacional com a introdução de uma álgebra baseada em ordem e implementaram essas conjecturas sobre o SGBD PostgreSQL, com uma extensão própria da linguagem SQL, priorizando o operador por ordenação. Esses trabalhos afirmam que devido ao custo dos operadores de consulta baseados em ordenação, eles devem ter prioridade no processamento, o que pode levar a resultados errôneos.

É importante caracterizar uma diferença fundamental entre álgebras e extensões de linguagens para tratar o conceito de ordem e aquelas voltadas ao tratamento de similaridade. Enquanto o conceito de *ranked-based queries* pressupõe a existência da ordem desassociada de uma consulta - as relações são mantidas ordenadas segundo um critério intrínseco à relação - as consultas por similaridade envolvem a ordem associada apenas à consulta - as tuplas apresentam uma ordem em relação ao centro de busca, sem a necessidade de um critério de ordem inerente aos dados.

Nota-se também que a ideia de ordem, apesar de normalmente estar associada à similaridade, não é inerente à similaridade. Por exemplo, seja um elemento s_i que é um dos k mais similares a um centro de consulta s_q mas não um dos k' mais similares ao mesmo centro, para $k' < k$. Como $k, k' \in \mathbb{N}$, existe uma ordem intrínseca entre k e k' , que pode ser aplicada nos correspondentes conjuntos-resposta obtidos pelo operador kNN . Mas o conceito de similaridade não depende que tal ordem exista.

2.6 Operadores físicos para junção

O processador de consultas de um SGBD é constituído de uma coleção de algoritmos e mecanismos que possibilitam a execução de expressões complexas usando múltiplos operadores, além de múltiplas ocorrências do mesmo operador. Analisados de um modo geral, o processador de consultas forma uma álgebra própria, que pode ser chamada de *álgebra física* de um SGBD, no sentido que todos os seus operadores recebem e produzem elementos que seguem a mesma estrutura, permitindo a reconfiguração dos módulos da melhor maneira para executar cada consulta.

A álgebra física é diferente, porém equivalente, à álgebra lógica do modelo de dados ou do SGBD. A álgebra lógica está relacionada com o modelo de dados e define como e quais consultas podem ser expressas nesse modelo. Por exemplo, a álgebra relacional é uma álgebra lógica. Por outro lado, a álgebra física é específica do gerenciador. Sistemas distintos podem implementar o mesmo modelo de dados e a mesma álgebra lógica, porém, podem fazer uso de álgebras físicas distintas. Por exemplo, um sistema pode realizar junções apenas através de laços aninhados, enquanto que outro sistema também implementa a técnica de intercalação (*merge*) e um terceiro gerenciador

permite realizar junções apenas através de *hash* [40, 41]. Outra diferença entre as álgebras física e lógica é que algoritmos e funções de custo são associados a operadores da álgebra física, nunca aos da álgebra lógica. Dessa forma, uma expressão da álgebra lógica deve ser mapeada para a álgebra física a fim de que possa ser executada. Entretanto, realizar o mapeamento entre a álgebra lógica e a física é uma tarefa complexa, envolvendo a escolha dos algoritmos e a ordem em que eles devem ser executados [22]. Além disso, os operadores das álgebras lógica e física nem sempre seguem o mapeamento um-para-um, como mostrado a seguir [40]:

- Múltiplos operadores lógicos podem ser implementados por um único operador físico: algoritmos de junção (físicos) permitem projetar o resultado diretamente (lógico: junção seguida de projeção).
- Determinados operadores físicos não existem na álgebra lógica: na álgebra relacional pura, que é uma álgebra de conjuntos, não existe um operador de ordenação (físico), uma vez que conjuntos não são ordenáveis por definição. Outro exemplo é operador físico que faz a leitura de uma relação em disco, que não existe na álgebra lógica.
- Propriedades dos operadores lógicos não se mantêm para os operadores físicos: os operandos dos operadores de união e intersecção (lógicos) podem ser livremente comutados, entretanto, algoritmos que processam essas operações (físicos) fazem distinção entre os operandos de entrada.

Os operadores físicos, também chamados de *métodos de acesso*, podem ser classificados como *unários* e *binários*, dependendo da quantidade de relações de entrada que eles manipulam. Nos SGBDs, é comum que um único operador físico implemente um “empacotamento” de vários operadores lógicos. Por exemplo, uma sequência dos operadores de seleção por um predicado seguida da projeção de alguns dos atributos será empacotada e executada completamente por um único operador físico (unário, por acessar apenas uma relação de entrada), chamado operador SP (*Select-Project*), como mostra a figura 2.4(a). A expressão algébrica executada por um operador SP corresponde à equação 2.3.

$$SP(l_1, c_1, R) \equiv \pi_{l_1}(\sigma_{c_1}(R)) \quad (2.3)$$

Por sua vez, uma sequência que envolve um operador de seleção por um predicado de busca sobre uma relação, a junção desse resultado com uma relação temporária gerada por outro operador físico, e uma projeção de atributos da relação resultante será empacotada e executada completamente por um único operador físico (binário, por acessar duas relações de entrada), chamado operador SPJ (*Select-Project-Join*), como mostra a figura 2.4(b). A expressão algébrica executada por um operador SPJ corresponde à equação 2.4.

$$SPJ(l_2, c_2, d_1, T, S) \equiv \pi_{l_2}\left(T \bowtie^{d_1}(\sigma_{c_2}(S))\right) \quad (2.4)$$

Os operadores SP e SPJ são assim chamados por corresponderem a maioria das consultas processadas pelos SGBDRs tradicionais. Todo operador físico, seja ele do tipo SP ou SPJ, é capaz de ler

Figura 2.4: Operadores lógicos encapsulados em operadores de acesso físico

apenas uma única relação-base de entrada. No caso de um operador SPJ (binário), ele deve sempre estar combinado a outro operador SPJ ou a um operador SP, sendo que estes proverão a relação R do operador SPJ. Assim, a expressão algébrica de um operador SPJ quando combinado a um operador SP corresponde à equação 2.5.

$$SPJ(l_2, c_2, d_1, SP(l_1, c_1, R), S) \equiv \pi_{l_2} \left(\left(\pi_{l_1} \left(\sigma_{c_1}(R) \right) \right) \bowtie^{d_1} \left(\sigma_{c_2}(S) \right) \right) \quad (2.5)$$

Muitos operadores físicos são implementados no modelo de um *iterador*, isto é, um agrupamento de três procedimentos que permitem ao consumidor do resultado de um operador físico obter uma tupla por vez. Os três métodos iteradores de um operador físico são [39]:

- *Open*: inicia o processo de obtenção de tuplas, mas não recupera tuplas. Este método inicializa as estruturas de dados necessárias à operação e invoca o método *Open* de outros operadores argumentos da operação.
- *GetNext*: retorna ao consumidor a próxima tupla do resultado e ajusta as estruturas de dados internas de modo a permitir a recuperação das tuplas subsequentes. A fim de obter mais tuplas para processar, esse método invoca o *GetNext* dos operadores passados como argumentos para o operador atual.
- *Close*: finaliza a execução depois que todas as tuplas requeridas pelo consumidor foram obtidas. Também invoca *Close* nos argumentos do operador.

Um dos principais operadores físicos existente nos SGBDRs é aquele que permite ler uma relação. Usualmente, esse operador é conhecido por *Table scan*, embora os nomes citados nesse texto possam variar entre implementações de sistemas específicos. O operador *Table scan* recupera todas as tuplas da relação dada como sua entrada. Além disso, esse operador empacota os operadores de seleção e projeção, de modo que cada tupla lida pode ser filtrada por um predicado de seleção e os atributos da projeção retornados pelo procedimento *GetNext* [39], e assim, corresponde a uma implementação de um operador SP. O operador *Table scan* não depende do uso de nenhum índice.

Um outro operador unário é o operador *Index scan*. Este é o operador básico para acessar uma relação que tenha um índice associado. Análogo ao *Table scan*, este operador empacota uma sequência de seleção e projeção e, portanto, é uma outra implementação de um operador SP. Porém, as condi-

Algoritmo 2.7: Nested-loop join

```

1 para cada tupla  $s \in S$  faça
2   para cada tupla  $r \in R$  faça
3     se  $r.a \theta s.b$  então
4       Concatena  $r$  e  $s$ ;
5       Adiciona  $\langle r, s \rangle$  no conjunto resposta;
6     fim
7   fim
8 fim

```

ções de seleção baseadas em atributos indexados são processadas fazendo uso do índice, enquanto que as demais condições são verificadas sequencialmente, a partir das tuplas obtidas pelo acesso ao índice [71]. Caso o índice contenha todos atributos a serem projetados pela consulta e nenhuma condição de seleção sobre outro atributo seja definida, não é necessário recuperar as tuplas da relação, de modo que o operador *Index-only scan* pode ser usado. Já no caso de um índice sozinho não conter todas as informações requeridas pela consulta mas existir outro índice que, associado ao primeiro, as contém, o operador *Index join* pode ser utilizado. Esse operador binário realiza a junção das informações de diversos índices.

A operação de junção é computacionalmente mais cara do que as seleções durante o processamento de consultas. Com relação aos operadores físicos que processam junção sobre relações, três deles são comumente implementados pelos SGBDRs: *Nested-loop join*, *Merge join* e *Hash join*.

O operador *Nested-loop join* (junção de laço aninhado) é o método mais básico de implementação de junção. Ele segue diretamente da definição da operação de junção. Uma das relações participantes da junção é designada como relação interna (*inner relation*) e a outra como relação externa (*outer relation*). Para cada tupla t da relação externa, todas as tuplas da relação interna são lidas e comparadas com a tupla t . Quando a condição de junção é satisfeita pelas tuplas analisadas, ambas são concatenadas e incluídas no conjunto resposta da junção. O algoritmo 2.7 mostra como processar $R \bowtie_{a\theta b} S$ como junção de laço aninhado. Este algoritmo apresenta custo quadrático em relação ao número de tuplas das duas relações.

No caso de os registros das relações participantes da junção estarem *fisicamente ordenados* com relação aos atributos de junção, a junção entre essas relações pode ser computada através de um operador mais eficiente chamado *Merge join*. Esse operador executa a junção em duas fases. Na primeira etapa, ambas as relações R e S são ordenadas pelos atributos a e b da condição de junção (*sort*). Então, ambas as relações são examinadas pela ordem dos atributos da junção, e as tuplas que satisfizerem a condição de junção são unidas para formar a relação resultado (*merge*). Quando as tuplas da primeira relação combinam com uma tupla da segunda relação, elas são concatenadas e colocadas na relação de resposta. O algoritmo 2.8 apresenta este processamento [32]. Caso as relações já estejam ordenadas, este algoritmo apresenta custo linear em relação ao número de tuplas das duas relações.

A eficiência do operador *Merge join* se deve ao fato de que ele reduz o número de comparações entre as tuplas. Devido à ordenação prévia, as tuplas da primeira relação não são comparadas com as tuplas da segunda relação que não satisfazem a condição de junção.

Algoritmo 2.8: (Sort-)Merge join

```

1 // Fase 1: ordenação
2 Ordena  $R$  sobre o atributo  $a$ ;
3 Ordena  $S$  sobre o atributo  $b$ ;
4 // Fase 2: intercalação
5  $i \leftarrow 1$ ;  $j \leftarrow 1$ ;
6 enquanto  $i \leq |R| \wedge j \leq |S|$  faça
7     se  $r_i.a > s_j.b$  então  $j \leftarrow j + 1$ ;
8     senão se  $r_i.a < s_j.b$  então  $i \leftarrow i + 1$ ;
9     senão
10         Concatena  $r_i$  e  $s_j$ ;
11         Adiciona  $\langle r_i, s_j \rangle$  no conjunto resposta;
12          $p \leftarrow j + 1$ ;
13         enquanto  $p \leq |S| \wedge r_i.a = s_p.b$  faça
14             Concatena  $r_i$  e  $s_p$ ;
15             Adiciona  $\langle r_i, s_p \rangle$  no conjunto resposta;
16              $p \leftarrow p + 1$ ;
17         fim
18          $q \leftarrow i + 1$ ;
19         enquanto  $q \leq |R| \wedge r_q.a = s_j.b$  faça
20             Concatena  $r_q$  e  $s_j$ ;
21             Adiciona  $\langle r_q, s_j \rangle$  no conjunto resposta;
22              $q \leftarrow q + 1$ ;
23         fim
24          $i \leftarrow q$ ;  $j \leftarrow p$ ;
25     fim
26 fim

```

Um outro operador físico, denotado por *Hash join*, alcança o mesmo resultado, mas de forma razoavelmente distinta. Na junção executada pelo operador *Hash join*, os valores dos atributos de junção da primeira relação serão usados por uma função de espalhamento (*hash*). Esses valores serão armazenados em uma tabela *hash*, onde cada entrada contém tuplas ou referências das tuplas da primeira relação. Dependendo da eficiência da função *hash*, mais de uma entrada podem ser mapeadas para a mesma posição. Então, para cada tupla da outra relação participante na junção, os valores dos atributos de junção são mapeados usando a mesma função *hash*. A seguir, os valores das tuplas mapeadas para a tabela *hash* são comparados com as tuplas “*haseadas*” da segunda relação. Se as tuplas comparadas satisfazem a condição de junção, elas são concatenadas para compor o resultado final. O algoritmo 2.9 apresenta esse processamento [39].

2.7 Considerações finais

Uma vez que os dados complexos não são organizados da maneira tradicionalmente feita pelos SGBDRs atuais e nem consultados através de comparações exatas, é necessário adotar novas técnicas para melhorar os processos de recuperação desses dados. Nesse contexto, este capítulo descreveu os aspectos teóricos mais importantes e úteis a esta tese com relação a consultas por

Algoritmo 2.9: Hash join

```

1 para cada tupla  $t_2 \in R_2$  faça
2   Computar a chave hash sobre  $t_2.b$ ;
3   Adicionar a chave na tabela hash;
4 fim
5 para cada tupla  $t_1 \in R_1$  faça
6   Computar a chave hash sobre  $t_1.a$ ;
7   se a chave recém computada coincide com uma posição ocupada na tabela hash então
8     se  $t_1.a \theta t_2.b$  então
9       Concatena  $t_1$  e  $t_2$ ;
10      Adiciona  $\langle t_1, t_2 \rangle$  no conjunto resposta;
11    fim
12  fim
13 fim

```

similaridade. Foram introduzidos os conceitos de espaço métrico, funções de distância, extratores de características, comparação por similaridade e operadores de busca por similaridade.

O modelo matemático mais adequado ao tratamento de dados complexos é o espaço métrico, que introduz medidas de proximidade entre os elementos. A noção de similaridade entre elementos é definida através de funções de distância ou de métricas. Existem distintos tipos métricas, capazes de relacionar elementos de mesma dimensionalidade, objetos adimensionais e conjuntos de elementos. As métricas permitem definir operadores de comparação por similaridade que podem ser empregados nos predicados de seleção e junção para gerar instâncias por similaridade de ambos os operadores.

Existem dois operadores de seleção e três operadores principais de junção por similaridade. Outras variações do operador de junção correspondem a combinações dos três tipos fundamentais. Esses cinco operadores mais descritos na literatura são sumarizados na tabela 2.4. Nessa tabela é possível notar a inexistência de um operador de seleção correspondente ao critério *k-closest neighbor* (proximidade). A ausência desse operador de seleção por similaridade é discutida pelo capítulo 4, constituindo uma das bases desta tese. O capítulo 3, apresenta uma das contribuições desta tese, o algoritmo *QuickNearest*, desenvolvido para agilizar junções por similaridade.

Tabela 2.4: Operadores de seleção e junção por similaridade

Operador de comparação	Seleção por similaridade	Junção por similaridade
<i>Similarity range</i>	$\sigma_{(S \text{ Rng}(d,\xi) s_q)} R$	$R \overset{(S_1 \text{ Rng}(d,\xi) S_2)}{\bowtie} S$
<i>k-nearest neighbor</i>	$\sigma_{(S \text{ kNN}(d,k) s_q)} R$	$R \overset{(S_1 \text{ kNN}(d,k) S_2)}{\bowtie} S$
<i>k-closest neighbor</i>	$\emptyset$	$R \overset{(S_1 \text{ kCN}(d,k) S_2)}{\bowtie} S$

QuickNearest

“Ecce nova facio omnia.”

Apocalypsis Ioannis 21,5

Este capítulo apresenta a primeira contribuição desta tese, o algoritmo *QuickNearest*, para executar o operador *k-nearest neighbor join*. Inicialmente, a seção 3.1 apresenta a definição formal desse operador. Ao contrário de um *range join*, o operador *k-nearest neighbor join* computa seu conjunto-resposta usando uma estratégia de contagem. Uma vez que esta característica influencia diretamente a forma como o operador prepara o conjunto-resposta, a seção 3.2 descreve as estruturas de dados e as operações sobre essas estruturas que são necessárias para computar o resultado final. A seguir, a seção 3.3 introduz os métodos genéricos responsáveis por computar uma junção por similaridade do tipo *kNN*, isto é, o algoritmo *QuickNearest* propriamente dito. Os algoritmos descritos nesta seção foram projetados com base no paradigma de divisão e conquista. Como o algoritmo genérico *QuickNearest* emprega métodos de particionamento de elementos no espaço como um de seus passos intermediários, as seções seguintes (3.4 e 3.5) descrevem as rotinas de apoio necessárias ao algoritmo para particionar o espaço de busca através das estratégias de hiperplano generalizado e por “bolas” métricas, respectivamente. A seção 3.6 apresenta a análise de complexidade de todo o algoritmo e a seção 3.7 descreve a avaliação experimental executada para avaliar o algoritmo *QuickNearest*. As considerações finais do capítulo são apresentadas na seção 3.8.

3.1 Definição do operador *k-nearest neighbor join*

Sejam R e S duas relações, $A_1 \subseteq R$ e $A_2 \subseteq S$ dois atributos complexos num domínio $\mathcal{S}$, d uma função de distância sobre $\mathcal{S}$ e $k \in \mathbb{N}^*$ uma constante inteira, positiva e não-nula. O objetivo central é computar um *k-nearest neighbor join* (*kNN join*) entre R e S . Formalmente, o operador *k-nearest neighbor join* é expresso de acordo com a definição 14.

Definição 14 (*k-nearest neighbor join*). Uma junção por similaridade com predicado do tipo *k-nearest neighbor*, $R \stackrel{(A_1 \text{ kNN}(d,k) A_2)}{\bowtie} S$, recupera um conjunto-resposta $T = \{t_1, t_2, \dots, t_p\}$, onde $p = k \cdot |R|$, tal que:

- $t_1 = \{\langle r, s \rangle \in (R \times S) \mid \forall \langle r, s' \rangle \in (R \times S) : d(r.A_1, s.A_2) < d(r.A_1, s'.A_2)\}$
- $t_2 = \{\langle r, s \rangle \in (R \times S) - \{t_1\} \mid \forall \langle r, s' \rangle \in (R \times S) - \{t_1\} : d(r.A_1, s.A_2) < d(r.A_1, s'.A_2)\}$
- $t_3 = \{\langle r, s \rangle \in (R \times S) - \{t_1, t_2\} \mid \forall \langle r, s' \rangle \in (R \times S) - \{t_1, t_2\} : d(r.A_1, s.A_2) < d(r.A_1, s'.A_2)\}$
- $t_4 = \{\langle r, s \rangle \in (R \times S) - \{t_1, t_2, t_3\} \mid \forall \langle r, s' \rangle \in (R \times S) - \{t_1, t_2, t_3\} : d(r.A_1, s.A_2) < d(r.A_1, s'.A_2)\}$
- ...
- $t_i = \left\{ \langle r, s \rangle \in (R \times S) - \left\{ \bigcup_{j=1}^{i-1} t_j \right\} \mid \forall \langle r, s' \rangle \in (R \times S) - \left\{ \bigcup_{j=1}^{i-1} t_j \right\} : d(r.A_1, s.A_2) < d(r.A_1, s'.A_2) \right\}$

Para fins de simplificação de notação, e sem perda de generalidade, os atributos complexos A_1 e A_2 , pertencentes às relações R e S , encontram-se omitidos no restante deste capítulo. Assim, a notação $d(r, s)$ refere-se, na verdade, à distância entre esses atributos nas tuplas r e s , ou seja, $d(r.A_1, s.A_2)$.

3.2 A estrutura do conjunto-resposta

Uma das etapas fundamentais do processamento de um *k-nearest neighbor join* está relacionada à forma como o algoritmo computa o conjunto-resposta. Esta etapa delineia como os pares obtidos no processo de junção são dispostos no conjunto-resposta e quais são as estruturas de dados utilizadas para armazenar essas tuplas.

Ao contrário de um *range join*, em que tão logo o predicado por similaridade expressando sua condição seja satisfeito as tuplas analisadas são pareadas e incluídas na resposta final, este mesmo comportamento não pode ser assumido como verdadeiro para uma junção por similaridade do tipo *kNN*.

Em um *k-nearest neighbor join*, duas tuplas podem vir a ser pareadas e incluídas no conjunto-resposta em uma das passagens do algoritmo, mas este mesmo par pode ser removido em uma iteração subsequente, de modo que ele não se faça presente na resposta final. A razão deste comportamento é explicada pelo fato de um operador do tipo *kNN* necessitar de controlar o tamanho do conjunto-resposta durante seu processamento, buscando por uma quantidade k de objetos. Assim, se o algoritmo encontra duas tuplas mais similares entre si do que algum outro par já presente no conjunto-resposta, o novo pareamento é realizado e o par de tuplas com elementos menos similares é substituído pelo par de maior similaridade. Para realizar esta etapa de substituição, os pares são comparados de acordo com a distância entre suas duas tuplas formadoras.

Portanto, uma operação *kNN* requer uma ordenação como uma operação implícita interna a seu processamento para que a operação de substituição de pares seja realizada. Essas duas operações (ordenação e substituição) não são executadas por um *range join*, sendo essa a principal razão

pela qual consultas envolvendo predicados do tipo *range* são, em geral, mais rápidas do que as consultas do tipo *kNN*.

Uma segunda consequência da dependência de operações de ordenação e substituição de pares é que um operador *kNN* é um operador bloqueante¹ na árvore de comandos de execução de uma consulta. Normalmente, todo SGBD prioriza a geração rápida de resultados e, de fato, isto pode ser alcançado com consultas do tipo *range*. Qualquer par que satisfaça um predicado *range* certamente faz parte da resposta e pode ser imediatamente mostrado ao usuário. Por outro lado, a mesma asserção não procede para as consultas *kNN*. Operadores com predicado *kNN* devem processar todo o conjunto de elementos antes de poder fornecer qualquer resposta, visto que substituições podem acontecer nas últimas iterações dos algoritmos. Assim, enquanto que um predicado do tipo *range* é baseado somente nos elementos analisados, um predicado *kNN* é baseado em toda a relação e a resposta final só é obtida após a relação ter sido completamente processada.

É importante destacar que a dependência de ordenação/substituição não caracteriza um problema de um operador do tipo *kNN*, seja ele seleção ou junção. Este tipo de comportamento é intrínseco ao operador. Entretanto, estas particularidades do processamento influenciam a definição das estruturas de dados responsáveis por armazenar a resposta da consulta, especialmente no caso do operador *k-nearest neighbor join*.

Operadores não bloqueantes, como o *range join*, processam os conjuntos-resposta de maneira que os novos pares encontrados são apenas adicionados. As estruturas de dados que compõem o conjunto-resposta, normalmente listas, têm na inserção sua principal operação: assim que um novo par satisfaz ao predicado da junção, ele é inserido no final da lista. Contudo isso não é suficiente para um operador *kNN* pois, por ser baseado em contagem, ele necessita de um conjunto maior de operações. Assim, para o operador *k-nearest neighbor join*, a estrutura de dados deve permitir no mínimo duas outras operações além da inserção: remoção, a fim de permitir substituições (remover um par menos similar e inserir outro mais similar), e ordenação, a fim de garantir a remoção do par mais dissimilar e manter a contagem.

A necessidade da ordenação é condição base para a operação de substituição. Nas etapas iniciais do processamento de um *kNN*, todos os pares encontrados são incluídos no conjunto-resposta. Assim que o conjunto-resposta contenha *k* pares, novos pares são adicionados apenas através da operação de substituição, onde o par menos similar presente no conjunto-resposta é substituído por outro mais similar. A procura pelo par de menor similaridade pode ser realizada de duas maneiras. A abordagem mais básica consiste em verificar cada par armazenado no resultado, isto é, realizar uma busca sequencial. De fato, no pior dos casos, esta operação é executada em tempo proporcional a $O(k)$, uma vez que a lista contém no máximo *k* elementos (os *k* vizinhos mais próximos). Após localizar o par $\langle r, s \rangle$ que tenha o maior valor de $d(r, s)$, ele pode ser substituído.

A outra abordagem consiste em manter o conjunto-resposta ordenado pelo valor da distância $d(r, s)$ de cada par. Neste caso, ao invés de armazenar um par, o conjunto resposta armazena uma tupla na forma $\langle r, s, \epsilon \rangle$, onde *r* e *s* são os objetos pertencentes a cada relação de junção e ϵ é o valor

¹Operador bloqueante: aquele que não é capaz de produzir qualquer resultado até que todas as tuplas de entrada tenham sido examinadas.

de dissimilaridade entre eles, computado de acordo com uma função de distância. O valor ϵ é armazenado para evitar cálculos de distância futuros, durante a etapa de substituição.

Manter o conjunto-resposta ordenado pelo valor de dissimilaridade entre os pares traz duas consequências diretas. A primeira é que o par de elementos de menor similaridade pode ser recuperado em tempo constante, proporcional a $O(1)$. Este par corresponde àquele presente no final da lista (ou no início, no caso de uma lista em ordem inversa). A segunda é que novos pares, ao serem inseridos, devem sê-lo nas posições corretas, de modo a respeitar a ordenação da lista. Embora a operação de inserção não seja mais realizada em tempo constante, o custo total do algoritmo de substituição ainda é menor que o custo da estratégia não ordenada. Através do uso de algoritmos como a busca binária, é possível localizar a posição de inserção de um novo par em tempo proporcional a $O(\lg k)$.

Devido às suas vantagens, a estratégia de manter o conjunto-resposta ordenado foi adotada na implementação do algoritmo *QuickNearest*. Contudo, existe, ainda, um artifício final na definição do conjunto-resposta do operador *k-nearest neighbor join*. Como pode ser observado na definição 14, uma junção $R \bowtie^{kNN} S$ recupera k tuplas de S para cada tupla presente em R . Portanto, cada tupla $r \in R$ deve possuir seu próprio conjunto de tuplas $s \in S$ com quem são pareadas, e a resposta final é dada pela união de cada um desses conjuntos individuais. Os conjuntos individuais devem ser mantidos separados, visto que para um par $\langle r, s_i \rangle$ ser substituído por outro par $\langle r, s_j \rangle$, eles são analisados com relação à mesma tupla r . Assim, ao contrário do que ocorre em um *k-closest neighbor join*, no *k-nearest neighbor join* cada tupla r é individualmente considerada.

Com relação à estrutura de dados que atende a esta necessidade de operações sobre o conjunto-resposta, pode-se empregar uma lista de tamanho $|R|$, onde cada nodo armazena a referência para o conjunto-resposta individual de uma tupla $r \in R$, ou seja, a referência para uma lista ordenada. Dessa forma, o conjunto-resposta final é constituído por uma lista de listas. Esta estrutura foi empregada nos algoritmos 3.2 e 3.4. Ambos realizam uma operação de junção *kNN* por laço aninhado como uma etapa intermediária do operador *QuickNearest*, como detalhado na seção 3.3.

3.3 O algoritmo *QuickNearest*

QuickNearest é um algoritmo que tem por objetivo computar um operador *k-nearest neighbor join*. O algoritmo foi desenvolvido para funcionar de modo independente de domínio, ou seja, ele é capaz de processar *kNN joins* sobre qualquer tipo de dado, seja multidimensional, adimensional ou puramente métrico. Para isto, o algoritmo requer apenas dois parâmetros: o valor da constante k , que é um parâmetro natural de um *kNN join*, e a indicação da função de distância usada.

Embora diversos métodos presentes na literatura (discutidos no capítulo 2) necessitem que a função de distância seja métrica, obedecendo à desigualdade triangular, esta propriedade não é um requisito para o algoritmo *QuickNearest*, visto que este algoritmo não emprega estratégias de indexação. Portanto, o algoritmo proposto é também capaz de computar *kNN joins* sobre dados não-métricos, cujas funções de distância atendem apenas as propriedades de reflexividade, simetria e não-negatividade.

Algoritmo 3.1: quickNearest(objs[], k, d(), result[])

Entrada: objs[]: o conjunto de tuplas de entrada.
 k: parâmetro da consulta.
 d(): a função de distância.
 result[]: a estrutura do conjunto-resposta.

```

1 se |objs| ≤ H então
2   selfNestedLoop(objs, k, d, result);
3   retorna;
4 fim
5 repita
6   pA ← pivot(objs);
7   pB ← pivot({objs - {pA}});
8   (partA, partB) ← partition(objs, pA, pB, d);
9 até |partA| ≥ k e |partB| ≥ k;
10 quickNearest(partA, k, d, result);
11 quickNearest(partB, k, d, result);
12 winDist ← windowDistance(partA, partB, pA, pB, d, result);
13 (winA, winB) ← computeWindows(partA, partB, pA, pB, winDist, d);
14 joinWindows(winA, winB, k, d, result);

```

QuickNearest é um algoritmo recursivo baseado na estratégia de divisão e conquista. Algoritmos deste tipo normalmente têm sua execução composta por três fases: divisão, conquista e combinação. Na primeira fase do *QuickNearest*, divisão, o problema de computar o kNN join é subdividido em problemas menores, ou seja, o espaço de busca é particionado em pequenos subespaços. Na próxima fase, conquista, o algoritmo executa kNN joins em cada subespaço. Como todos os subespaços produzidos contêm poucos elementos, o kNN é computado usando o método de laço aninhado. Por fim, na fase de combinação, as soluções individuais encontradas em cada subespaço são agrupadas e submetidas a uma verificação final que produzirá a resposta do kNN join. As três etapas do *QuickNearest* são mostradas no algoritmo 3.1, que executa uma junção de uma relação consigo mesma: $R \overset{kNN}{\bowtie} R$. O caso geral, $R \overset{kNN}{\bowtie} S$, pode ser obtido invocando diretamente o algoritmo 3.3, posteriormente apresentado.

O algoritmo 3.1 recebe quatro parâmetros de entrada. O primeiro é a relação a ser processada, contendo as tuplas para a junção: *objs*. Como esta versão do algoritmo *QuickNearest* executa em memória primária, o parâmetro *objs* é definido como uma variável composta homogênea unidimensional, o que permite acelerar o desempenho da operação. O segundo parâmetro é o valor k da consulta. O terceiro parâmetro é a função de distância empregada para computar a dissimilaridade entre os pares de elementos de *objs*. A passagem da função de distância como um dos parâmetros assegura a característica de processamento genérico para o qual o algoritmo *QuickNearest* foi desenvolvido. Uma vez que o cálculo do valor da distância entre os elementos é a única operação necessária para comparação de elementos, este parâmetro pode ser qualquer função de distância adequada para o tipo de dado a ser comparado. Por fim, o quarto parâmetro é uma referência para a estrutura de dados do conjunto-resposta, como descrito na seção 3.2.

O algoritmo *QuickNearest* inicia seu processamento verificando se a cardinalidade do conjunto de dados é menor que uma constante H predefinida, chamada de *limiar de sublimação* (*sublimation limit*). Quando existem muitos dados, assume-se que é interessante investir tempo na estruturação dos dados, e procura-se particionar os dados em regiões, idealmente com a mesma quantidade de elementos, de maneira que as operações de busca possam explorar todas as regiões que não contribuirão para o resultado. Quando a quantidade de dados é pequena, o custo de estruturar os dados aumenta, e os dados são tratados sem estruturação. A caracterização do limiar de sublimação H é feita na seção 3.3.2. Assim, quando a cardinalidade dos dados é menor do que H , o problema é resolvido pela estratégia de força bruta e o algoritmo se encerra. Estas operações são mostradas nos passos 1-4 do algoritmo 3.1. Neste ponto do *QuickNearest* é aplicada uma das regras da estratégia de divisão e conquista: se a instância do problema é suficientemente pequena, basta resolvê-la através do método de força bruta. O método de força bruta empregado para computar o kNN *join* é o algoritmo de laço aninhado (*nested loop*), invocado pelo passo 2 do algoritmo 3.1. O pseudocódigo correspondente a esta rotina é apresentado pelo algoritmo 3.2.

O algoritmo 3.2 recebe os mesmos parâmetros de entrada da função principal do *QuickNearest* (algoritmo 3.1). Os laços aninhados estão declarados nos passos 1-2. Eles são responsáveis por percorrer todas as tuplas de entrada em *objs* e adicionar os k pares mais similares entre si no conjunto-resposta. Inicialmente, a condição expressa no passo 3 verifica se as tuplas analisadas não são as mesmas. No caso de ambas serem distintas, testa-se se o conjunto-resposta associado à tupla do laço externo de junção possui menos que k pares, e, em caso positivo, as tuplas analisadas no laço aninhado são combinadas e inseridas no conjunto-resposta (passos 4-5). Caso contrário, o conjunto-resposta já possui k pares e novas inserções ocorrem apenas por meio de substituição, como descrito na seção 3.2. Para isso, o passo 6 verifica se o conjunto-resposta contém algum par de similaridade menor que as tuplas analisadas, e, em caso afirmativo, o par menos similar é removido no passo 7 e o novo par inserido pelo passo 8. Caso contrário, nenhuma ação é realizada, visto que os pares armazenados no conjunto-resposta já são mais similares entre si que o par de tuplas analisado pela iteração atual do laço.

Na sequência de processamento do algoritmo 3.1, no caso do conjunto de dados possuir mais elementos que o necessário para satisfazer a condição do passo 1, a operação de laço aninhado não é realizada. Neste caso, o *QuickNearest* particiona os elementos do espaço de busca em conjuntos de dados menores e, recursivamente, invoca a si próprio para processar cada partição gerada.

Para realizar a divisão dos elementos do espaço de busca, o algoritmo *QuickNearest* pode utilizar duas estratégias. A primeira realiza o particionamento através de um hiperplano, objetivando repartir o conjunto de dados em duas porções, uma de cada lado do hiperplano. O segundo método realiza o assim chamado particionamento por bolas métricas. Ele define uma fronteira no espaço de busca de modo a separar a região do espaço que está a até uma distância máxima de um objeto do fixo. Assim, os elementos são separados em dois conjuntos: o conjunto dos elementos que estão dentro da região de fronteira e o conjunto dos elementos que estão além da fronteira. Os detalhes de execução de ambas as estratégias de particionamento, bem como as particularidades de cada uma, são explicados pelas seções 3.4 e 3.5, respectivamente, para os métodos de hiperplano e divisão por bolas.

Algoritmo 3.2: selfNestedLoop(objs[], k, d(), result[])

Entrada: objs[]: o conjunto de tuplas de entrada.
 k: parâmetro da consulta.
 d(): a função de distância.
 result[]: a estrutura do conjunto-resposta.

```

1 para cada  $r \in \text{objs}$  faça
2   para cada  $s \in \text{objs}$  faça
3     se  $r \neq s$  então
4       se  $|\text{result}[r]| < k$  então
5          $\text{result}[r] \leftarrow \text{result}[r] \cup \{ \langle r, s \rangle \};$ 
6       senão se  $d(r, s) < \max_{\langle a, b \rangle \in \text{result}[r]} d(a, b)$  então
7          $\text{result}[r] \leftarrow \text{result}[r] - \{ \text{argmax}_{\langle a, b \rangle \in \text{result}[r]} d(a, b) \};$ 
8          $\text{result}[r] \leftarrow \text{result}[r] \cup \{ \langle r, s \rangle \};$ 
9       fim
10    fim
11  fim
12 fim

```

Para realizar o particionamento dos dados, nos passos 6-7, o algoritmo 3.1 seleciona dois elementos de referência (pivôs) pA e pB dentre os elementos do conjunto de dados. Os pivôs são usados para definir o hiperplano ou para delimitar a região de fronteira (bola) no espaço de busca. A única condição imperativa é que os pivôs sejam elementos distintos. Devido à sua simplicidade, não é fornecida a implementação do método `pivot()`, invocado nos passos 6-7: apenas se escolhe e retorna um elemento do espaço de busca de maneira aleatória. No passo 8, o conjunto de dados *objs* é dividido em duas partições: *partA* e *partB*.

Depois de particionar o espaço de busca, o algoritmo 3.1 é recursivamente invocado nos passos 10-11. Cada chamada recursiva recebe como conjunto de entrada uma das partições do conjunto de dados original, isto é, *partA* e *partB*. A cada chamada recursiva realizada, as cardinalidades dos conjuntos *partA* e *partB* são sempre menores que a cardinalidade do conjunto de dados recebido como entrada e, por consequência, a operação de laço aninhado do passo 2 eventualmente será executada na recursão. Quando a última chamada recursiva do algoritmo 3.1 é completada, o conjunto-resposta contém uma aproximação do resultado real do *kNN join*. Para diversas aplicações, esta resposta aproximada pode ser suficiente e o algoritmo pode ser finalizado neste ponto. Contudo, o interesse desta tese é obter a resposta exata.

3.3.1 Resposta aproximada $\times$ resposta exata

Sempre que k for menor ou igual à cardinalidade da relação S , assume-se que o conjunto-resposta aproximado contém $k \cdot |\text{objs}|$ pares de elementos, e esta assertiva é sempre verdadeira, conforme mostrado adiante. O problema é que pares relevantes podem não ter sido encontrados pelo algoritmo e foram substituídos por pares de menor similaridade, levando à aproximação do resultado.

A perda de pares relevantes acontece na etapa de particionamento. Quando o conjunto de entrada *objs* é dividido em dois subconjuntos *partA* e *partB*, algumas tuplas de *partA* podem ser combinadas

(a) Um hiperplano generalizado divide os elementos em duas partições A e B. Para $k = 2$, o hiperplano separa elementos que deveriam ser combinados, tais como os pares $\langle e1, e3 \rangle$ and $\langle e5, e1 \rangle$. Este problema é solucionado computando e combinando as janelas das partições.

(b) As janelas (windows) A e B contêm os elementos das partições A e B que podem vir a ser pareados na operação de junção. A distância entre as janelas define a extensão das janelas a partir do hiperplano.

Figura 3.1: Particionamento do espaço de busca por hiperplano generalizado

com outras tuplas presentes em $partB$, e estes pares podem vir a compor a resposta final, como ilustrado na figura 3.1(a). Uma vez que o algoritmo 3.2 não chega a parear as duas partições $partA$ e $partB$ conjuntamente, processando-as apenas separadamente, é necessário que o algoritmo 3.1 realize a verificação desses conjuntos, a fim de obter a resposta exata da operação de junção kNN .

A figura 3.1(a) mostra a situação de perda de pares, onde o espaço de busca é dividido em duas partições, uma de cada lado de um hiperplano. O hiperplano, por sua vez, é definido pelos pivôs pA e pB . Sem perda de generalidade e visando a intuitividade do método, a figura 3.1(a) ilustra um espaço bidimensional, onde a métrica L_2 foi usada em uma junção 2-NN. Ambos os elementos $e2$ e $e4$ são combinados com o elemento $e1$. Já o elemento $e5$ é pareado com os elementos $e3$ e $e6$. Entretanto, os pares verdadeiramente corretos deveriam ser $\langle e1, e3 \rangle$ ao invés de $\langle e1, e4 \rangle$, e $\langle e5, e1 \rangle$ no lugar de $\langle e5, e6 \rangle$.

De modo similar ao algoritmo *Quickjoin* (seção 2.5), que computa *range joins*, o algoritmo *QuickNearest* cria duas partições adicionais, chamadas de janelas (windows), que contêm as tuplas da partição $partA$ que podem vir a ser pareadas com as tuplas da outra partição $partB$.

O algoritmo *Quickjoin* é capaz de computar as janelas facilmente. De fato, esta operação pode ser realizada diretamente, como um passo adicional do algoritmo 2.4 (passos 3-4), visto que, para *range joins*, as janelas dependem apenas do limiar de similaridade ξ . Entretanto, o algoritmo *QuickNearest* não recebe outro parâmetro de consulta além do valor da constante k , e esta constante não pode ser usada para estimar um limiar de similaridade porque não é possível obter tais estimativas para *kNN joins*, como discutido na seção 2.5.

Para computar suas janelas, é necessário que o algoritmo *QuickNearest* primeiro determine o quão longe os objetos das partições $partA$ e $partB$ podem estar ou do hiperplano ou da fronteira da bola métrica que os separa para ainda serem incluídos nas respectivas janelas. Este valor é chamado de *distância de "janelamento"* (window distance). Ele é calculado tomando como referência a dissimilaridade do k -ésimo par do conjunto-resposta. Entretanto, não é uma regra que a *window*

Algoritmo 3.3: joinWindows(objsA[], objsB[], k, d(), result[])**Entrada:** objsA[], objsB[]: os conjuntos de tuplas de entrada.

k: parâmetro da consulta.

d(): a função de distância.

result[]: a estrutura do conjunto-resposta.

```

1 se  $|objsA| + |objsB| \leq H$  então
2   nestedLoop(objsA, objsB, k, d, result);
3   retorna;
4 fim
5  $pA \leftarrow \text{pivot}(\{objsA \cup objsB\})$ ;
6  $pB \leftarrow \text{pivot}(\{objsA \cup objsB\} - \{pA\})$ ;
7  $(partA1, partA2) \leftarrow \text{partition}(objsA, pA, pB, d)$ ;
8  $(partB1, partB2) \leftarrow \text{partition}(objsB, pA, pB, d)$ ;
9 joinWindows(partA1, partB1, k, d, result);
10 joinWindows(partA2, partB2, k, d, result);
11  $wd1 \leftarrow \text{windowDistance}(partA1, partA2, pA, pB, d, result)$ ;
12  $wd2 \leftarrow \text{windowDistance}(partB1, partB2, pA, pB, d, result)$ ;
13  $winDist \leftarrow \max(wd1, wd2)$ ;
14  $(winA1, winA2) \leftarrow \text{computeWindows}(partA1, partA2, pA, pB, winDist, d)$ ;
15  $(winB1, winB2) \leftarrow \text{computeWindows}(partB1, partB2, pA, pB, winDist, d)$ ;
16 joinWindows(winA1, winB2, k, d, result);
17 joinWindows(winA2, winB1, k, d, result);

```

distance seja a distância do k -ésimo par de elementos, visto que essa k -ésima distância pode não englobar elementos da outra partição.

Como a *window distance* é obtida de modo distinto para cada estratégia de particionamento, os detalhes de processamento e critérios de poda de cada método são detalhados nas seções 3.4 e 3.5 para os particionamentos por hiperplano e por bolas, respectivamente. Independente da estratégia de particionamento escolhida, o algoritmo 3.1 computa a distância das janelas no passo 12, invocando o método `windowDistance()`.

Uma vez computado, o valor de *window distance* é utilizado no passo 13 do algoritmo 3.1 para distribuir alguns dos elementos dos conjuntos *partA* e *partB* nas janelas *winA* e *winB*, respectivamente, como ilustrado pela figura 3.1(b). A função `computeWindows()`, chamada no passo 13, é responsável por analisar os elementos de cada partição e incluí-los nas janelas apropriadas. Do mesmo modo que no passo 12, existem condições distintas a serem satisfeitas para que um elemento seja inserido em uma janela. Estas condições também dependem da estratégia de particionamento adotada e são descritas nas seções 3.4 e 3.5 para os particionamentos por hiperplano e por bolas, respectivamente.

O passo final do algoritmo 3.1 consiste em computar o *kNN join* sobre os elementos das janelas *winA* e *winB*. Esta operação substituirá os pares menos relevantes por pares mais similares, levando à resposta exata do algoritmo. Para isto, o método `joinWindows()` é invocado no passo 14. O pseudocódigo que detalha este método é apresentado no algoritmo 3.3.

Algoritmo 3.4: nestedLoop(objsA[], objsB[], k, d(), result[])**Entrada:** objsA[], objsB[]: os conjuntos de tuplas de entrada.

k: parâmetro da consulta.

d(): a função de distância.

result[]: a estrutura do conjunto-resposta.

```

1  para cada  $r \in \text{objsA}$  faça
2      para cada  $s \in \text{objsB}$  faça
3          se  $|\text{result}[r]| < k$  então
4               $\text{result}[r] \leftarrow \text{result}[r] \cup \{ \langle r, s \rangle \};$ 
5          senão se  $d(r, s) < \max_{\langle a, b \rangle \in \text{result}[r]} d(a, b)$  então
6               $\text{result}[r] \leftarrow \text{result}[r] - \{ \text{argmax}_{\langle a, b \rangle \in \text{result}[r]} d(a, b) \};$ 
7               $\text{result}[r] \leftarrow \text{result}[r] \cup \{ \langle r, s \rangle \};$ 
8          fim
9      se  $|\text{result}[s]| < k$  então
10          $\text{result}[s] \leftarrow \text{result}[s] \cup \{ \langle s, r \rangle \};$ 
11         senão se  $d(s, r) < \max_{\langle a, b \rangle \in \text{result}[s]} d(a, b)$  então
12              $\text{result}[s] \leftarrow \text{result}[s] - \{ \text{argmax}_{\langle a, b \rangle \in \text{result}[s]} d(a, b) \};$ 
13              $\text{result}[s] \leftarrow \text{result}[s] \cup \{ \langle s, r \rangle \};$ 
14         fim
15     fim
16 fim

```

O algoritmo 3.3 é semelhante ao algoritmo 3.1. Ele recebe: os dois conjuntos de entrada *objsA* e *objsB* a serem processados, e que correspondem às janelas *winA* e *winB* do algoritmo 3.1; o parâmetro de consulta *k*; a função de distância empregada e a referência para a estrutura de dados que compõe o conjunto-resposta. O algoritmo 3.3 segue a mesma estratégia de divisão e conquista empregada no algoritmo 3.1. No caso dos conjuntos de entrada possuírem mais elementos do que o limiar de sublimação previamente estabelecido, eles são particionados em conjuntos menores e o mesmo método é invocado de maneira recursiva para processar cada partição separadamente.

O algoritmo 3.3 inicia verificando se a opção de processar a junção usando a estratégia de laço aninhado é computacionalmente adequada. No caso da quantidade de elementos ser relativamente pequena, o que é determinado pelo limiar de sublimação *H* no passo 1, o passo 2 também invoca um método de laço aninhado (algoritmo 3.4), mas que é distinto daquele anteriormente descrito (algoritmo 3.2).

Esta segunda versão do algoritmo de laço aninhado (algoritmo 3.4) tem como objetivo verificar possíveis combinações de tuplas das partições que foram anteriormente separadas. Assim, este método combina as tuplas do conjunto de entrada *objsA* com as tuplas do conjunto de entrada *objsB* e vice-versa, mas não pareia as tuplas pertencentes ao mesmo conjunto. No algoritmo 3.4, os passos 1-2 iteram sobre ambos os conjuntos. Nos passos 3-8, o método verifica se as tuplas de *objsB* devem ser combinados com a tupla de *objsA* analisada no laço externo. Os passos 9-14 realizam esta mesma verificação, mas considerando as tuplas de *objsA* em relação às tuplas de *objsB*. Esta verificação dupla permite que o algoritmo 3.4 compare as tuplas de ambos os conjuntos em uma única passada pelo laço aninhado. De modo análogo aos passos 4-9 do algoritmo 3.2, os passos 3-8

Figura 3.2: Particionamento na invocação recursiva do método `joinWindows()`: os conjuntos anteriores *winA* e *winB*, recebidos como *objsA* e *objsB*, são divididos, respectivamente, nas partições A1/A2 (em vermelho) e B1/B2 (azul), separadas pelo novo hiperplano. Depois que as janelas são geradas, a verificação final ocorre entre os conjuntos A1/B2 e A2/B1.

e 9-14 do algoritmo 3.4 também controlam a cardinalidade do conjunto-resposta correspondente a cada tupla analisada pelo laço.

No caso da soma da quantidade de elementos das janelas *objsA* e *objsB* ser maior que o limiar de sublimação *H*, o algoritmo 3.3 subdivide os elementos do espaço de busca em conjuntos menores. Nos passos 5-6, são selecionados dois pivôs *pA* e *pB* distintos para que, no passo 7, eles sejam usados para dividir o conjunto *objsA* nas duas partições *partA1* e *partA2*. A mesma operação de divisão é realizada para os elementos de *objsB*. O passo 8 os divide nos subconjuntos *partB1* e *partB2*. Como o algoritmo 3.3 recebe dois conjuntos de entrada, o método `partition()` deve ser invocado duas vezes, para dividir cada conjunto. O método `partition()` é o mesmo empregado no algoritmo 3.1, onde os dados são particionados usando as estratégias de hiperplano generalizado ou particionamento por bolas métricas.

Posteriormente, os passos 9-10 do algoritmo 3.3 realizam a chamada recursiva do método `joinWindows()`, a fim de processar as tuplas da partição *partA1* que podem vir a ser pareadas com as tuplas da partição *partB1*, bem como as tuplas de *partA2* que têm chance de combinação com as tuplas de *partB2*. Nos passos 9-10, a maior parte dos pares falso positivos incluídos no conjunto-resposta pelo algoritmo 3.2 são substituídos por pares mais similares que devem, de fato, compor o resultado final.

Como cada etapa de particionamento necessita que novas janelas sejam geradas, alguns poucos pares falso positivos ainda podem ser encontrados no conjunto-resposta. Para encontrar e eliminar esses pares, os passos 13-14 do algoritmo 3.3 calculam uma nova *window distance* e quatro novas janelas são geradas nos passos 14-15. Duas dessas janelas, *winA1* e *winA2*, são obtidas pelo particionamento do conjunto *objsA*, e as outras duas janelas, *winB1* e *winB2*, correspondem à divisão do conjunto *objsB*. Estas janelas podem ser visualizadas na figura 3.2. Uma vez que o conjunto-resposta é modificado em cada chamada recursiva do método, o valor da *window distance* decresce sucessivamente. Dessa forma, esse valor é sempre computado novamente (passos 11-13) a fim de reduzir a quantidade de tuplas nos conjuntos de janelas, o que contribui para evitar comparações desnecessárias e assim acelerar o algoritmo.

Uma vez que o algoritmo 3.3 verificou possíveis combinações entre os conjuntos *objsA* e *objsB*, entre as partições *partA1/partA2*, e entre as partições *partB1/partB2*, a última etapa de verificação é realizada sobre as novas janelas, mostradas na figura 3.2. Assim, nos passos 16-17, o algoritmo 3.3 realiza a checagem das tuplas dispostas na janela *winA1* com as tuplas de *winB2*, e das tuplas em *winA2* com as tuplas de *winB1*.

Ao término da última invocação recursiva do método `joinWindows()`, o algoritmo *QuickNearest* finaliza seu processamento. Os pares menos similares previamente incluídos no conjunto-resposta já foram substituídos por pares mais similares, haja visto que o algoritmo processou todas as subdivisões do espaço de busca, e, portanto, a resposta *exata* e *final* do *kNN join* foi encontrada.

O algoritmo 3.1 depende de dois detalhes importantes, que são mandatórios para sua correta operação.

Após executar os passos 10-11, o algoritmo 3.1 terá obtido uma aproximação da resposta final. Mesmo que as janelas ainda não tenham sido geradas, esta resposta aproximada contém a quantidade de pares esperada: $k \cdot |objs|$. Neste ponto, o algoritmo 3.1 atinge enorme ganho de desempenho uma vez que o processo de análise das janelas refina a resposta aproximada na resposta exata através de substituições. Já as substituições acontecem analisando regiões específicas do espaço de busca, isto é, apenas as tuplas que têm probabilidade de ser incluídas no conjunto-resposta são processadas. No raro caso de não ocorrer nenhuma substituição, o resultado computado ao fim dos passos 10-11 torna-se final e contém a cardinalidade total de um *kNN join*.

O algoritmo 3.1 garante que a cardinalidade do conjunto-resposta é mantida controlando o tamanho das partições geradas no passo 9. Em cada chamada recursiva, o algoritmo 3.1 processa uma das partições obtidas na invocação anterior. Como as cardinalidades das partições tornam-se menores e um laço aninhado é invocado no passo 2, o algoritmo 3.2 deve garantir, de algum modo, que a cardinalidade do conjunto-resposta seja preservada, uma vez que ele é o único método que manipula a estrutura de dados do resultado.

A única forma do algoritmo 3.2 garantir tal comportamento é receber um conjunto de entrada contendo ao menos as k tuplas necessárias à operação *kNN*. Dessa forma, na computação de um laço aninhado sobre um conjunto de k tuplas, é garantido que cada tupla do conjunto seja pareada com outras k tuplas, tendo em vista que o algoritmo tem ao menos k possibilidades de combinação.

Por esta razão, o conjunto-resposta obtido neste ponto do algoritmo é uma aproximação, uma vez que os elementos com maior similaridade entre si podem estar dispostos em partições opostas. Assim, as partições *partA* e *partB* que foram obtidas pelo algoritmo 3.1 necessitam ter ao menos k elementos, justificando a existência da estrutura de laço *repita*—até nos passos 5-9.

3.3.2 O limiar de sublimação H

O parâmetro de consulta k , que é usado no algoritmo 3.1 para definir o tamanho mínimo das partições, é também empregado para determinar o valor do limiar de sublimação H , usado no passo 1 dos algoritmos 3.1 e 3.3, que especifica a quantidade limite de tuplas para a execução de um laço aninhado ao invés do particionamento do espaço de busca.

Para que o algoritmo *QuickNearest* realize chamadas recursivas e obtenha desempenho devido à estratégia de divisão e conquista, o valor do limiar de sublimação deve ser menor que a cardinalidade do conjunto de entrada. Caso contrário, uma junção por laço aninhado é imediatamente executada. Portanto, como pode ser facilmente observado, o valor $|objs| - 1$ torna-se o limite superior para o limiar de sublimação H .

O problema, entretanto, está em como definir um limite inferior para H . Analisando o algoritmo 3.1, percebe-se que escolher um valor $H < k$ torna-se problemático. Como o algoritmo 3.1 divide o espaço de busca, alguma de suas invocações recursivas iniciará com um conjunto de entrada $objs$ em que $H < |objs| < k$. Neste caso, a condição indicada no passo 1 não é satisfeita, e o algoritmo passa para a fase de particionamento dos dados (passos 5-9). Como $|objs| < k$, não é possível dividir o conjunto $objs$ em duas partições contendo ao menos k elementos cada uma. Assim, o algoritmo 3.1 entraria em um laço infinito, tentando em vão reparticionar o espaço.

Por exemplo, seja uma situação em que $H = 8$, $k = 10$ e $|objs| = 15$. Se o algoritmo 3.1 dividir o conjunto de entrada $objs$ de modo que $|partA| = 9$ e $|partB| = 6$, a próxima chamada recursiva terá $H = 8$, $k = 10$ e $|objs| = 9$. Nesta configuração, observa-se que a condição do passo 1 não é satisfeita. Além disso, o conjunto $objs$ não pode ser dividido em dois subconjuntos contendo ao menos 10 elementos cada um, levando ao laço infinito. Outros valores tais como $H = k$ ou $H = k + 1$ são igualmente problemáticos, como pode ser constatado repetindo essa análise.

Quando o algoritmo 3.1 particiona o espaço de busca, cada partição individual deve conter ao menos k elementos, condição necessária para garantir a cardinalidade do conjunto-resposta, como anteriormente mencionado. Se tanto $|partA|$ quanto $|partB|$ são no mínimo k , a cardinalidade do conjunto $objs$ que será dividido em duas partes deve ser, obrigatoriamente, no mínimo $2k$. Assim, o limite inferior para o limiar de sublimação H pode ser estabelecido em $2k - 1$: para conjuntos de entrada onde $|objs| \geq 2k$, é possível realizar o particionamento em dois conjuntos com ao menos k elementos; para conjuntos menores que $2k$, um laço aninhado é invocado. Isto não significa, contudo, que um valor $H = 2k - 1$ ou que valores próximos a $2k - 1$ sejam uma boa escolha para o algoritmo *QuickNearest*.

Por exemplo, seja uma configuração onde o limite inferior para H seja $2k - 1$. Esta escolha satisfaz a condição na qual cada partição deve conter ao menos k elementos. Assim que $|objs| \leq 2k - 1$, o algoritmo se reduz a um laço aninhado, a chamada recursiva se encerra e os $2k - 1$ elementos não prendem o algoritmo em um laço infinito nos passos 5-9. O problema é uma chamada recursiva onde $objs$ inicia contendo uma cardinalidade igual ou pouca coisa maior que $2k$ elementos e o conjunto deve ser particionado. Como os passos 6-7 escolhem pivôs de forma estocástica, não há garantias de que os pivôs pA e pB retornados produzirão uma divisão de elementos balanceada, onde ambas as partições $partA$ e $partB$ contenham ao menos k elementos. Dessa forma, o laço dos passos 5-9 poderá ser executado diversas vezes até que uma escolha de pivôs produza uma divisão adequada do conjunto $objs$, onde $|partA| = |partB| = k$.

Como artifício, o algoritmo 3.1 deve sempre receber como entrada um conjunto com cardinalidade suficiente para evitar a dependência de uma escolha de pivôs perfeita. Infelizmente, não há valores exatos para tal estimativa. Analisando o algoritmo 3.1, se H receber um valor $i \cdot k$, sendo $i \geq 3$,

o método `pivot()` recebe uma quantidade maior de elementos e a probabilidade de executar diversas iterações diminui.

Ao contrário do algoritmo 3.1, o algoritmo 3.3 não requer um controle de cardinalidade semelhante. Uma vez que este último algoritmo manipula um conjunto-resposta que já contém $k \cdot |objs|$ pares, não é necessário que as partições contenham ao menos k elementos. Assim, este método apenas aplica a estratégia de particionamento, sem a necessidade de verificar pela cardinalidade das partições geradas. Com relação à geração das partições, elas ocorrem através das estratégias de hiperplano generalizado e particionamento por bolas, descritas nas respectivas seções 3.4 e 3.5.

3.4 Estratégia de particionamento por hiperplano generalizado

O particionamento do conjunto de dados é uma das etapas mais importantes do algoritmo *QuickNearest*. De fato, a cada chamada recursiva, o método `partition()`, responsável por realizar a divisão dos elementos, é invocado uma vez pelo algoritmo 3.1 e duas vezes pelo algoritmo 3.3.

A operação de particionamento está relacionada com a fase de divisão da estratégia de divisão e conquista, e traz melhorias significativas ao desempenho do algoritmo *QuickNearest*. A principal ideia é realizar divisões recursivas dos elementos do espaço de busca em subconjuntos menores, até que os subconjuntos contendo poucas tuplas possam ser eficientemente processados pela estratégia de laço aninhado. Para isto, o método `partition()` executa de modo similar ao bem conhecido algoritmo *Quicksort*, o que também é realizado pelo algoritmo *Quickjoin* original.

Entre as várias formas de separar elementos dispostos no espaço métrico, o particionamento por hiperplano generalizado é uma das mais empregadas. Usando esta estratégia, o algoritmo *QuickNearest* também adota um hiperplano generalizado, onde o conjunto de dados é dividido em duas partições. Inicialmente, o algoritmo *QuickNearest* escolhe dois pivôs de modo aleatório e então atribui os elementos do espaço de busca que estão mais próximos do primeiro pivô à primeira partição e os elementos mais próximos do segundo pivô à segunda partição, onde a relação de proximidade entre os elementos é dada pela função de distância. Elementos que estejam à mesma distância de ambos os pivôs podem ser atribuídos a qualquer uma das partições, porém, sempre de modo determinístico.

No caso do algoritmo *QuickNearest*, não é esperado que o particionamento usando hiperplano divida o conjunto de dados em dois subconjuntos balanceados. Embora uma configuração como esta permitiria ao algoritmo processar ainda mais rapidamente, o algoritmo 3.1 é capaz de manipular partições desbalanceadas, sendo que este é o comportamento mais observado na prática. A versão do método `partition()` usando um hiperplano generalizado é apresentada pelo algoritmo 3.5.

O algoritmo 3.5 recebe quatro parâmetros de entrada. O primeiro é o conjunto de dados a ser subdividido. Os dois próximos parâmetros são os pivôs pA e pB , escolhidos aleatoriamente pelos algoritmos 3.1 ou 3.3, e, por fim, a função de distância utilizada. De modo análogo aos algoritmos 3.1 e 3.3, o método `partition()` assume que o conjunto de dados é representado por uma variável composta homogênea unidimensional, o que permite realizar movimentações de objetos *in loco*.

Algoritmo 3.5: partition(objs[], pA, pB, d()) - Hiperplano**Entrada:** objs[]: o conjunto de tuplas de entrada.

pA, pB: elementos de referência (pivôs).

d(): a função de distância.

Saída : Os subconjuntos *partA* e *partB* contendo a divisão das tuplas de *objs*.

```

1  start ← 0;
2  end ← |objs| - 1;
3  enquanto verdadeiro faça
4      enquanto start < end faça
5          se d(objs[end], pA) < d(objs[end], pB) então
6              pausa;
7          fim
8          end ← end - 1;
9      fim
10     enquanto start < end faça
11         se d(objs[start], pB) ≤ d(objs[start], pA) então
12             pausa;
13         fim
14         start ← start + 1;
15     fim
16     se start < end então
17         objs[start] ↔ objs[end];           // troca os objetos
18         start ← start + 1;
19         end ← end - 1;
20     senão
21         pausa;
22     fim
23 fim
24 se start = end então
25     se d(objs[start], pB) ≤ d(objs[start], pA) então
26         end ← end - 1;
27     fim
28 fim
29 partA ← objs[0 : end];
30 partB ← objs[end + 1 : |objs| - 1];
31 retorna (partA, partB);

```

Os dois primeiros passos do algoritmo 3.5 inicializam as posições inicial e final dos elementos do conjunto de dados. O laço presente no passo 3 é responsável por percorrer os elementos recebidos. Da mesma forma que o algoritmo *Quicksort*, que redistribui os elementos de acordo com sua comparação com o pivô, a função *partition()* executa de modo similar, mantendo os elementos que estão mais próximos ao pivô *pA* do lado oposto a todos os outros elementos que são mais próximos do pivô *pB*.

Para realizar a distribuição dos dados, o laço dos passos 4-9 analisa os elementos das posições finais do conjunto *objs*. Enquanto todos os elementos estiverem corretamente posicionados, isto é, mais próximos do pivô *pB* que do pivô *pA*, nenhuma ação é realizada. O laço somente é interrompido ao encontrar um elemento que seja mais próximo do pivô *pA* e, portanto, deveria estar posicionado na

outra partição. Na sequência, o laço dos passos 10-15 realiza a mesma verificação do laço anterior, porém, inicia pelos elementos da primeira metade do conjunto *objs* e compara suas distâncias com cada pivô, até que um elemento que seja mais próximo a *pB* seja identificado. Quando este segundo laço é interrompido, o algoritmo encontrou dois elementos erroneamente dispostos e, então, realiza sua permutação nos passos 16-19.

O algoritmo 3.5 continua a verificação dos elementos de *objs* até que o último elemento, aquele que divide o conjunto ao meio, seja alcançado. A procura pela partição à qual o último elemento pertence é realizada nos passos 24-28. Finalmente, no passo 31, o algoritmo 3.5 devolve os dois subconjuntos *partA* e *partB* contendo a divisão dos elementos de *objs*.

Embora a escolha aleatória dos pivôs *pA* e *pB* na maioria das vezes produza partições desbalanceadas, elas nunca estarão vazias. Como cada pivô é também um dos elementos do conjunto de entrada *objs*, sua distância para si mesmo (zero) é sempre menor que sua distância para o outro pivô. Portanto, cada partição possui, seguramente, ao menos um elemento.

O método `partition()` permite explorar duas oportunidades de otimização durante todo o processamento do algoritmo *QuickNearest*. A primeira é aplicar técnicas de programação dinâmica a fim de reduzir a quantidade de cálculos de distância. Como as distâncias de todos os elementos para cada pivô não variam, uma vez que os elementos são fixos no espaço de busca, os valores pré-computados nos passos 5-11 podem ser disponibilizados, visando evitar a repetição de cálculos. A segunda otimização é obtida ao manipular referências de objetos na memória, ao invés dos objetos reais, o que evita a realização de cópias nas passagem de parâmetro de cada método invocado.

Depois que as partições *partA* e *partB* são recursivamente analisadas pelo algoritmo 3.1 ou 3.3, o passo seguinte verifica quais elementos que estão dispostos em partições opostas devem ser pareados para compor o resultado final. Analisar cada par de elementos separadamente, sendo um elemento de cada partição, e verificar se o par deve ou não ser incluído no conjunto-resposta é o mesmo que executar um laço aninhado. Tal forma de processamento não é considerada, visto que adicionaria um custo quadrático ao algoritmo *QuickNearest*. Além disso, muitos dos pares verificados não seriam incluídos no resultado.

Para combinar os elementos de partições distintas, a estratégia adotada é então, para cada partição, estabelecer uma região limítrofe contendo apenas os elementos que têm possibilidade de pareamento, isto é, delimitar também aqui uma janela. Para isto, é necessário descobrir até quão distante do hiperplano estas janelas devem se estender, ou seja, computar a *window distance*.

Em um primeiro momento, é possível verificar os pares do conjunto-resposta e escolher como distância aquela correspondente ao par mais dissimilar. Denotando tal distância por d_k , ela poderia ser usada como valor de *window distance*. Assim, em uma junção *kNN*, d_k corresponde à distância entre alguma *k*-ésima tupla *s* pareada com alguma outra tupla *r*. Como o par $\langle r, s \rangle$ pertence à mesma partição, se algum outro elemento *t* da partição oposta puder ser combinado com *r*, a distância $d(r, t)$ será, obrigatoriamente, menor que d_k , como mostrado pela figura 3.3(a). No caso de $d(r, t) \geq d_k$, não há substituição no conjunto-resposta, e o elemento *r* está pareado com os *k* elementos *s* mais similares.

(a) A distância $d_k = d(e1, e4)$ (*window distance*) permite substituir pares menos similares, como $\langle e1, e4 \rangle$, por mais similares, como $\langle e1, e3 \rangle$.

(b) A distância d_k não ultrapassa a fronteira do hiperplano. Assim, a distância entre $e3$ e $e4$ é escolhida como *window distance*, por ser aquela que cruza o hiperplano.

Figura 3.3: Definição da *window distance*

O problema em escolher d_k como valor do parâmetro *window distance* é que as janelas computadas terão mais elementos do que o realmente necessário, resultando em comparações desnecessárias. O par $\langle r, s \rangle$ tal que o valor da dissimilaridade $d(r, s) = d_k$ pode estar localizado em uma região do espaço métrico em que a distância d_k , tomada a partir de r , nunca se aproxima do hiperplano. Assim, embora d_k seja a maior distância do conjunto-resposta, seu alcance máximo quando centrada em um elemento r ainda permanece dentro da mesma partição, como mostrado pela figura 3.3(b).

Neste contexto, o cálculo de valor da *window distance* requer que uma verificação adicional seja realizada. Entre as k distâncias de cada elemento r que compõe o conjunto-resposta, somente são consideradas as k -ésimas distâncias cuja extensão a partir de r seja capaz de alcançar regiões além da fronteira delimitada pelo hiperplano. Deste conjunto de distâncias válidas, a maior delas é a que deve ser selecionada como *window distance*.

Assim, entre as distâncias cujos pares são possíveis candidatos a sofrerem substituição no conjunto-resposta, é selecionada a maior distância, visto que ela caracteriza um limite superior ao valor da *window distance*. Uma vez que a junção dos elementos nas janelas é também processada pelo laço aninhado do algoritmo 3.4, este primeiro critério de poda reduz a quantidade de comparações de elementos, otimizando o desempenho do algoritmo *QuickNearest*. O método que computa o valor da *window distance* para a estratégia de hiperplano generalizado é apresentado no algoritmo 3.6.

O algoritmo 3.6 recebe como parâmetros de entrada as partições previamente computadas, os dois pivôs que geraram o hiperplano, a função de distância empregada e a referência para a estrutura de dados do conjunto-resposta. Em seu passo 1, o algoritmo 3.6 supõe que nenhuma operação de substituição ocorrerá e, assim, como não haverá criação de janelas, o valor da *window distance* é zero. De fato, se as partições são criadas de modo que não haja substituições, o comportamento esperado do algoritmo é que não existam janelas para serem processadas. O laço dos passos 2-6 analisa os elementos do conjunto $objsB$, procurando pela k -ésima distância dos elementos de $objsB$ que fazem parte do conjunto-resposta e tal que o valor da distância extrapole os limites do hiperplano. Como verificado no passo 4, se o elemento não satisfizer a condição de cruzamento do hiperplano, seu valor de distância não é considerado. Caso contrário, no passo 5, a *window distance* é atualizada para o maior valor encontrado até aquela iteração, ou seja, seu limite superior. Os passos 8-13

Algoritmo 3.6: `windowDistance(objsA[], objsB[], pA, pB, d(), result[])` - Hiperp.**Entrada:** `objsA[], objsB[]`: os conjuntos de tuplas de entrada.`pA, pB`: elementos de referência (pivôs).`d()`: a função de distância.`result[]`: a estrutura do conjunto-resposta.**Saída** : A k -ésima maior distância que cruza o hiperplano.

```

1 winDist ← 0;
2 para cada  $s \in \text{objsB}$  faça
3    $\text{dist} \leftarrow \max_{\langle a,b \rangle \in \text{result}[s]} d(a, b);$ 
4   se  $d(s, pB) + \text{dist} > d(s, pA) - \text{dist}$  então // critério de poda
5      $\text{winDist} \leftarrow \max(\text{winDist}, \text{dist});$ 
6   fim
7 fim
8 para cada  $r \in \text{objsA}$  faça
9    $\text{dist} \leftarrow \max_{\langle a,b \rangle \in \text{result}[r]} d(a, b);$ 
10  se  $d(r, pA) + \text{dist} \geq d(r, pB) - \text{dist}$  então // critério de poda
11     $\text{winDist} \leftarrow \max(\text{winDist}, \text{dist});$ 
12  fim
13 fim
14 retorna winDist;

```

executam o mesmo tipo de processamento, agora com relação aos elementos do conjunto *objsA*. Quando este último laço se encerra, o algoritmo devolve o valor da *window distance*.

Assim que a *window distance* é computada, os algoritmos 3.1 e 3.3 podem invocar o método `computeWindows()` para criar as janelas. Os elementos de ambas as partições são verificados e aqueles que estão dispostos na extensão da *window distance*, a partir do hiperplano, são incluídos nas janelas, para processamento futuro.

Como o hiperplano não pode ser expresso analiticamente ao se trabalhar com espaços métricos, a distância de um elemento e para um hiperplano nem sempre pode ser definida explicitamente. Por exemplo, se os dados são representados na forma de vetores n -dimensionais e a função de distância empregada for a Euclidiana, é possível definir a distância exata de qualquer elemento para um hiperplano. Porém, no caso de dados puramente métricos, como sequências de caracteres, não é possível representar um hiperplano, mas ainda é possível dividir os elementos em dois conjuntos tomando como base suas distâncias para pivôs. Portanto, a distância de um elemento para um hiperplano pode não ser exata, mas, ainda assim, é possível estimar um limite inferior para o seu valor. A expressão estima o valor da distância de um elemento arbitrário e para um hiperplano é apresentada pela equação 3.1 (demonstração no apêndice A), onde pA e pB são os pivôs.

$$d_{Hip} = \frac{d(e, pA) - d(e, pB)}{2} \quad (3.1)$$

Como a equação 3.1 não computa a distância real, mas apenas o limite inferior do valor real, as janelas *winA* e *winB* geradas pelo método `computeWindows()` ainda podem conter alguns poucos elementos extras, que nunca serão pareados, porém verificados. Assim, embora ainda se gaste algum tempo de processamento com esses falso-positivos, garante-se que não existem falsos-

Algoritmo 3.7: computeWindows(objsA[], objsB[], pA, pB, winDist, d()) – Hiperp.**Entrada:** objsA[], objsB[]: os conjuntos de tuplas de entrada.

pA, pB: elementos de referência (pivôs).

winDist: a distância das janelas.

d(): a função de distância.

Saída : As janelas *winA* and *winB*.

```

1 winA ← ∅;
2 winB ← ∅;
3 para cada s ∈ objsB faça
4     se  $\frac{d(s, pA) - d(s, pB)}{2} \leq winDist$  então
5         | winB ← winB ∪ {s};
6     fim
7 fim
8 para cada r ∈ objsA faça
9     se  $\frac{d(r, pA) - d(r, pB)}{2} \geq (-1) * winDist$  então
10        | winA ← winA ∪ {r};
11    fim
12 fim
13 retorna (winA, winB);

```

negativos e portanto a resposta obtida é exata. O algoritmo 3.7 apresenta a versão do método computeWindows() baseada em hiperplanos generalizados.

O algoritmo 3.7 recebe como parâmetros de entrada as duas partições, ambos os pivôs que geraram o hiperplano, a distância das janelas e a função de distância empregada nas comparações. De modo similar ao método windowDistance(), o algoritmo 3.7 percorre ambos os conjuntos de dados e inclui os elementos que satisfazem à equação 3.1 (passos 4 e 9) em suas respectivas janelas: os elementos do conjunto *objsA* são alocados na janela *winA* e os elementos da partição *objsB* são alocados na janela *winB*. O laço dos passos 3-7 verifica os elementos do conjunto *objsB*, e o laço dos passos 8-12 verifica os elementos do conjunto *objsA*.

Com relação ao critério de poda, a análise dos elementos de *objsA* retorna um valor negativo, visto que os elementos estão mais próximos do pivô *pA* e suas distâncias para *pB* são maiores. Visando resolver esse impasse (funções de distâncias não devolvem valores negativos), o segundo membro da equação 3.1 é multiplicado por -1 e o comparador relacional do passo 9 é invertido. Outra possibilidade seria considerar o valor absoluto do referido termo. Estas questões, contudo, não se fazem presentes na estratégia de particionamento por bolas. Além disso, esta estratégia não necessita ter um limite inferior para a distância das janelas. Como explanado na próxima seção 3.5, o particionamento por bolas computa valores exatos.

3.5 Estratégia de particionamento por bolas

A versão do algoritmo QuickNearest baseada em particionamento por bolas ao invés de divisão por hiperplano generalizado é a técnica mais rápida, visto que o particionamento dos dados faz uso

Figura 3.4: Particionamento do espaço de busca por bolas métricas

apenas de valores exatos. Ou seja, as distâncias empregadas no processo de divisão dos elementos do espaço de busca não caracterizam limites inferiores, e sim os seus valores reais. Assim, a operação de laço aninhado entre as janelas produzidas pelo algoritmo 3.3 processa menos pares quando o particionamento por bolas é utilizado do que na estratégia de divisão por hiperplano, justificando o seu desempenho superior, como mostrado na avaliação experimental da seção 3.7.

Da mesma forma que na divisão por hiperplano, a estratégia de particionamento por bolas também emprega métodos para dividir o conjunto de dados, calcular a *window distance* e gerar as janelas propriamente ditas. Contudo, estes métodos apresentam modificações para permitir a separação de acordo com a segunda estratégia. À exceção destes detalhes, as versões dos algoritmos 3.1, 3.2, 3.3 e 3.4 anteriormente apresentadas permanecem as mesmas, visto que elas não dependem da estratégia de particionamento de dados.

A divisão dos elementos do conjunto de dados é a primeira operação relacionada à estratégia de particionamento. Para a técnica de particionamento por bolas, o algoritmo *QuickNearest* usa dois elementos de referência (pivôs) para delimitar uma região do espaço de busca. Os dois pivôs são selecionados de maneira aleatória. O primeiro pivô assume o papel de ser o centroide da região delimitada. O segundo pivô é usado para definir a fronteira da região de separação, e para isto, é usada a distância entre os dois pivôs para determinar o alcance máximo da fronteira. Dessa forma, o espaço de busca fica dividido em duas partições.

A primeira partição é composta por todos os elementos cuja distância para o centroide é menor ou igual que a distância do centroide para a fronteira, ou seja, para o segundo pivô. Já a segunda partição é formada pelos elementos cuja distância para o centroide é maior que esse valor. A figura 3.4 ilustra este tipo de divisão de acordo com os pivôs pA (centroide) e pB , bem como as janelas obtidas pela execução do algoritmo, também definidas dentro e fora da região delimitada.

O método `partition()` adaptado para realizar a divisão dos elementos do espaço seguindo a estratégia de particionamento por bolas é apresentado pelo algoritmo 3.8.

O algoritmo 3.8 recebe como parâmetros de entrada o conjunto de elementos, os dois pivôs responsáveis pela criação da fronteira da região fachada (bola) e a função de distância empregada nas comparações. Semelhante ao seu homólogo por hiperplano generalizado, o algoritmo 3.8 começa definindo os limites da estrutura do conjunto de dados nos passos 1-2. Na sequência, o algoritmo percorre a estrutura do vetor em ambas as direções e realiza a troca de elementos. Todos os elemen-

Algoritmo 3.8: `partition(objs[], pA, pB, d())` - Bolas**Entrada:** `objs[]`: o conjunto de tuplas de entrada.`pA`, `pB`: elementos de referência (pivôs).`d()`: a função de distância.**Saída** : Os subconjuntos *partA* e *partB* contendo a divisão das tuplas de *objs*.

```

1  start  $\leftarrow$  0;
2  end  $\leftarrow$   $|objs| - 1$ ;
3  enquanto verdadeiro faça
4      enquanto start < end faça
5          se  $d(objs[end], pA) < d(pA, pB)$  então
6              pausa;
7          fim
8          end  $\leftarrow$  end - 1;
9      fim
10     enquanto start < end faça
11         se  $d(objs[start], pA) \geq d(pA, pB)$  então
12             pausa;
13         fim
14         start  $\leftarrow$  start + 1;
15     fim
16     se start < end então
17          $objs[start] \leftrightarrow objs[end]$ ; // troca os objetos
18         start  $\leftarrow$  start + 1;
19         end  $\leftarrow$  end - 1;
20     senão
21         pausa;
22     fim
23 fim
24 se start = end então
25     se  $d(objs[start], pA) \geq d(pA, pB)$  então
26         end  $\leftarrow$  end - 1;
27     fim
28 fim
29 partA  $\leftarrow$  objs[0 : end];
30 partB  $\leftarrow$  objs[end + 1 :  $|objs| - 1$ ];
31 retorna (partA, partB);

```

tos dispostos dentro da região de fronteira são posicionados de um lado (esquerdo, na estrutura), enquanto que todos os elementos além da fronteira são dispostos na partição oposta. Para isto, o laço dos passos 4-9 itera sobre os elementos que estão além da bola até que um elemento localizado em seu interior seja encontrado. Então, o laço dos passos 10-15 realiza a mesma verificação, até encontrar um elemento localizado fora da bola. Para redistribuir os elementos erroneamente localizados em suas partições corretas, os passos 16-19 fazem a respectiva permutação.

O processo de verificação continua até que todos os elementos do conjunto de entrada *objs* tenham sido analisados. Com relação ao elemento final, sua pertinência a alguma das duas partições é computada pelos passos 24-28. Ao final da execução do algoritmo 3.8, o conjunto de dados *objs* está particionado em dois subconjuntos: *partA* e *partB*.

Algoritmo 3.9: windowDistance(objsA[], objsB[], pA, pB, d(), result[]) - Bolas**Entrada:** objsA[], objsB[]: os conjuntos de tuplas de entrada.

pA, pB: elementos de referência (pivôs).

d(): a função de distância.

result[]: a estrutura do conjunto-resposta.

Saída : A k -ésima maior distância que cruza a fronteira.

```

1 winDist ← 0;
2 para cada s ∈ objsB faça
3   dist ← max(a,b) ∈ result[s] d(a, b);
4   se d(s, pA) − dist ≤ d(pA, pB) então // critério de poda
5     winDist ← max(winDist, dist);
6   fim
7 fim
8 para cada r ∈ objsA faça
9   dist ← max(a,b) ∈ result[r] d(a, b);
10  se d(r, pA) + dist > d(pA, pB) então // critério de poda
11    winDist ← max(winDist, dist);
12  fim
13 fim
14 retorna winDist;
```

Novamente, ambas as partições computadas não necessitam de serem balanceadas. A versão baseada em particionamento por bolas do algoritmo *QuickNearest* também processa partições apresentando cardinalidades díspares. No entanto, o algoritmo 3.8 garante que cada partição contenha ao menos um único elemento. Por exemplo, no caso de todos os elementos do conjunto de dados serem atribuídos para a partição interna à fronteira, o último elemento, aquele que gerou a distância até a fronteira (pB), estará localizado exatamente sobre a fronteira. No algoritmo 3.8, este elemento é atribuído à segunda partição, como pode ser verificado nos passos 11 e 25, onde a condição também verifica a relação de igualdade.

De maneira análoga ao particionamento por hiperplano generalizado, as partições obtidas pela estratégia de divisão por bolas podem conter elementos que devem ser combinados, embora estejam dispostos em partições opostas. Para identificar esses pares e incluí-los no conjunto-resposta, são realizadas operações similares àquelas descritas na seção 3.4. O algoritmo *QuickNearest* computa o valor do parâmetro *window distance* e cria as janelas.

A *window distance* é obtida verificando-se cada elemento do conjunto-resposta e comparando se a distância dos pares na k -ésima posição permite atingir regiões que cruzam a fronteira definida pelos pivôs. Este cálculo é realizado pela versão por bolas métricas do método windowDistance(), apresentado no algoritmo 3.9.

O algoritmo 3.9 recebe como parâmetros de entrada as duas partições, os dois pivôs, a função de distância e a referência para a estrutura de dados do conjunto-resposta. O laço dos passos 2-7 analisa os elementos localizados além da fronteira delimitada pelos pivôs. Para cada par $\langle r, s \rangle$ que compõe o conjunto-resposta, o método verifica (passo 4) se a k -ésima distância tomada a partir de r ultrapassa a fronteira definida, e assim, permite alcançar os elementos internos à bola. Um laço análogo é visto nos passos 8-13, mas com relação aos elementos que estão dentro da fronteira. O

Algoritmo 3.10: computeWindows(objsA[], objsB[], pA, pB, winDist, d()) – Bolas**Entrada:** objsA[], objsB[]: os conjuntos de tuplas de entrada.

pA, pB: elementos de referência (pivôs).

winDist: a distância das janelas.

d(): a função de distância.

Saída : As janelas *winA* and *winB*.

```

1 winA ← ∅;
2 winB ← ∅;

3 para cada s ∈ objsB faça
4   se d(pA, pB) + winDist ≥ d(s, pA) então
5     winB ← winB ∪ {s};
6   fim
7 fim

8 para cada r ∈ objsA faça
9   se d(r, pA) + winDist ≥ d(pA, pB) então
10    winA ← winA ∪ {r};
11  fim
12 fim
13 retorna (winA, winB);

```

algoritmo procura pela k -ésima maior distância que atinge regiões além da fronteira, armazenando seu valor. O passo 10 realiza a checagem dos elementos nesta situação. Os pares que não satisfazem a esta condição são descartados da análise, visto que suas distâncias não alcançam a outra partição. Ao final do processamento, o algoritmo 3.9 devolve o valor da *window distance*.

Após obter o valor da *window distance*, as janelas podem ser criadas através da chamada do método computeWindows(). Na estratégia de particionamento por bolas, as janelas são delimitadas por duas outras fronteiras radiais àquela definida pelos pivôs: uma menor e uma maior. A fronteira menor delimita uma região interna à bola tal que sua distância para a fronteira dos pivôs é o valor da *window distance*. A fronteira maior engloba a região externa à fronteira dos pivôs, seguindo o mesmo raciocínio. Ambas as janelas e suas fronteiras podem ser vistas na figura 3.4. O algoritmo 3.10 introduz a versão de particionamento por bolas do método computeWindows().

A execução do algoritmo 3.10 é semelhante à execução do algoritmo 3.7. Ele recebe como entrada as duas partições, os dois pivôs, o valor da *window distance* e a função de distância. Após inicializar os conjuntos das janelas como vazios nos passos 1-2, o laço dos passos 3-7 percorre os elementos da partição externa e inclui no conjunto *winB* todos aqueles que estão dentro da área definida para esta janela. Para ser incluído na janela, o elemento deve satisfazer a condição expressa no passo 4. Na sequência, o laço dos passos 8-12 realiza a verificação dos elementos dispostos dentro da bola. Caso estes elementos satisfaçam a condição definida no passo 9, eles são incluídos na janela *winA*.

No caso do algoritmo 3.10, as janelas contêm apenas os elementos que, seguramente, estão dentro da distância estabelecida a partir da fronteira dos pivôs. Na sequência, a verificação de quais elementos serão pareados e ocasionarão substituições no conjunto-resposta é realizada no momento de processar a junção das janelas, de acordo com o algoritmo 3.3 (joinWindows()), anteriormente descrito.

3.6 Análise do algoritmo *QuickNearest*

Um dos fatores que causa maior influência no desempenho do algoritmo *QuickNearest* é a forma como o conjunto de dados é particionado. O melhor caso do algoritmo ocorre quando, em cada invocação recursiva, os elementos são divididos igualmente entre as duas partições, ou seja, quando ocorre um particionamento balanceado. De modo similar, o pior caso ocorre quando, em cada chamada recursiva, o particionamento é realizado de forma que uma das partições contém um único elemento (o pivô) e a outra contém o restante do conjunto de dados. Neste caso, o algoritmo torna-se assintoticamente mais lento. Portanto, o desempenho do algoritmo *QuickNearest* depende da escolha dos pivôs usados para definir o hiperplano generalizado ou a região de fronteira do particionamento por bolas.

Embora o pseudocódigo do método `pivot()`, invocado pelos algoritmos 3.1 e 3.3, não tenha sido apresentado, sua implementação é fundamentalmente básica. Uma vez que o método apenas devolve um elemento de modo aleatório entre os elementos recebidos como entrada, ele executa em tempo proporcional a $\Theta(1)$.

A cada chamada recursiva, o algoritmo 3.1 executa o método `partition()` ao menos uma vez, além de realizar duas chamadas recursivas de si mesmo e invocar os métodos `computeWindows()` e `joinWindows()` uma vez. A função `partition()`, considerando tanto a estratégia de hiperplano generalizado quanto a de particionamento por bolas, realiza uma única varredura do conjunto de dados de entrada. Como em cada invocação ela recebe uma fração ρ do conjunto de dados, esta função consome tempo proporcional a $O(\rho n)$, tal que n representa o tamanho do conjunto de dados de entrada.

Analogamente à função `partition()`, os métodos `windowDistance()` e `computeWindows()` percorrem os elementos do conjunto de entrada apenas uma vez, independente da estratégia de particionamento de dados adotada. A cada chamada recursiva, ambos também recebem como entrada uma fração da quantidade de elementos do conjunto de dados. Portanto, suas complexidades de execução também são dadas por $O(\rho n)$.

Os métodos não-recursivos restantes, `selfNestedLoop()` (algoritmo 3.2) e `nestedLoop()` (algoritmo 3.4), podem ser facilmente verificados como apresentando complexidade temporal proporcional a $O(\rho^2 n^2)$.

Dessa forma, o custo total do algoritmo *QuickNearest* ao processar um conjunto de dados contendo n elementos é expresso de acordo com a equação 3.2.

$$QN(n) = \lambda \cdot (P(n) + QN(\rho n) + QN((1 - \rho)n) + JW(\omega_1 \rho n, \omega_2 (1 - \rho)n) + f(\rho n)) + (1 - \lambda) \cdot NL(\rho^2 n^2) \quad (3.2)$$

Na equação 3.2, QN é o custo do método `quickNearest()`, P denota uma chamada ao método `partition()`, JW representa a invocação do método `joinWindows()`, f é o custo agregado de

ambos os métodos `windowDistance()` e `computeWindows()`, e NL representa ambos os métodos `selfNestedLoop()` e `nestedLoop()`.

Ainda na equação 3.2, o método `quickNearest()` é sempre invocado recebendo n elementos, isto é, o conjunto de dados completo. A função `partition()` divide esses n elementos em dois conjuntos contendo, respectivamente, ρn e $(1 - \rho)n$ elementos, sendo que esses conjuntos são as entradas da próximas duas invocações recursivas de `quickNearest()`. Uma vez que ambas as estratégias de particionamento descritas nas seções 3.4 e 3.5 impedem que as partições sejam vazias, a fração ρ está dentro do intervalo aberto $]0, 1[$. Na sequência, o método `joinWindows()`, denotado por JW , recebe uma fração ω da quantidade de objetos de cada partição, ou seja, dois conjuntos contendo $\omega_1 \rho n$ e $\omega_2 (1 - \rho)n$ elementos. Como o tamanho das janelas é normalmente desigual, os valores ω_1 e ω_2 não são necessariamente iguais. O custo linear em termos de ρn de ambos os métodos `windowDistance()` e `computeWindows()` encontra-se combinado em f . Por fim, NL denota a invocação de um dos algoritmos que processam o laço aninhado. A cada nível da recursão, o algoritmo pode tanto executar um laço aninhado ou então particionar o conjunto de dados. O valor $\lambda = 0$ representa a ocorrência de um laço aninhado, enquanto que um valor $\lambda = 1$ indica uma divisão dos elementos do espaço de busca.

A expressão que computa o desempenho da função `joinWindows()`, dada pela equação 3.3, é obtida de modo similar à da equação 3.2, sendo que m e n são os tamanhos das partições recebidas como entrada e ω_i é uma fração da quantidade de elementos da janela i .

$$JW(m, n) = \lambda \left(P(m) + P(n) + JW(\rho m, \rho n) + JW((1 - \rho)m, (1 - \rho)n) + f(\rho m) + f(\rho n) + \right. \\ \left. JW(\omega_1 \rho m, \omega_2 (1 - \rho)n) + JW(\omega_3 (1 - \rho)m, \omega_4 \rho n) \right) + (1 - \lambda)NL(\rho^2 n^2) \quad (3.3)$$

As equações 3.2 e 3.3 expressam o custo do algoritmo *QuickNearest* em cada nível de recursão. A figura 3.5 ilustra a invocação de cada método recursivo, a cada nível. Como pode ser notado nesta figura, o custo total do algoritmo depende da profundidade da recursão.

Considerando uma árvore de recursão com ℓ níveis, no nível zero há apenas uma invocação para o método principal `quickNearest()`. No nível 1, a chamada anterior de `quickNearest()` se divide em três operações recursivas, que irão compor o custo total do deste nível. Na figura 3.5, os métodos não-recursivos descritos anteriormente estão omitidos, visto que eles não influenciam a profundidade da árvore de recursão. O custo do algoritmo *QuickNearest*, no nível 1, é expresso pela equação 3.4.

$$QN_1(n) = P(n) + QN(\rho n) + QN((1 - \rho)n) + JW(\omega_1 \rho n, \omega_2 (1 - \rho)n) + f(\rho n) \quad (3.4)$$

Na equação 3.4, pode-se observar que o custo é dado por uma invocação do método `partition()`, duas invocações recursivas de `quickNearest()`, e uma de `joinWindows()`, acrescidos do custo combinado em f das funções `windowDistance()` e `computeWindows()`.

Figura 3.5: Árvore de recursão: QN e JW representam chamadas para os métodos `quickNearest()` e `joinWindows()`. A cada invocação, o método `quickNearest()` é executado duas vezes e `joinWindows()` uma vez.

De maneira análoga ao nível 1, no segundo nível da recursão cada chamada de `quickNearest()` é novamente subdividida, e o método `joinWindows()`, executado no nível 1, produz quatro invocações de si próprio no nível 2. Desta forma, realizando as substituições das equações 3.2 e 3.3 na equação 3.4, tem-se que o custo do algoritmo *QuickNearest* no segundo nível da recursão é expresso pela equação 3.5.

$$\begin{aligned}
 QN_2(n) = & P(n) + \left(P(\rho n) + QN(\rho^2 n) + QN(\rho(1-\rho)n) + f(\rho^2 n) \right) + JW(\omega_1 \rho^2 n, \omega_2 \rho(1-\rho)n) + \\
 & \left(P((1-\rho)n) + QN(\rho(1-\rho)n) + QN((1-\rho)^2 n) + JW(\omega_1 \rho(1-\rho)n, \omega_2(1-\rho)^2 n) + \right. \\
 & \left. f(\rho(1-\rho)n) \right) + \left(P(\omega_1 \rho n) + P(\omega_2(1-\rho)n) + JW(\omega_1 \rho^2 n, \rho \omega_2(1-\rho)n) + \right. \\
 & JW((1-\rho)\omega_1 \rho n, (1-\rho)\omega_2(1-\rho)n) + f(\omega_1 \rho^2 n) + f(\omega_2 \rho(1-\rho)n) + \\
 & \left. JW(\omega_1^2 \rho^2 n, \omega_2^2(1-\rho)^2 n) + JW(\omega_1 \omega_3(1-\rho)\rho n, \omega_2 \omega_4 \rho(1-\rho)n) \right)
 \end{aligned} \tag{3.5}$$

De forma similar, é possível estimar facilmente o custo do algoritmo *QuickNearest* para os níveis subsequentes da recursão. Observando a figura 3.5, pode-se notar que, a cada nível, a quantidade de invocações de QN dobra. Assim, para um nível arbitrário i , tem-se que QN é executado 2^i vezes. Por sua vez, os JW são gerados tanto pelos JW do nível anterior como pelos QN. Cada JW produz quatro invocações no nível posterior, enquanto que cada QN gera outro JW no próximo nível da árvore. Ainda na figura 3.5, em seu nível 1, há apenas um JW. No nível 2, este JW produz um grupo de quatro JW, enquanto que os dois QN do nível 1 produzem dois grupos com um JW, totalizando seis JW. No nível 3, tem-se um grupo de 16 JW (produzidos pelos 4 JW do nível 2), dois grupos com quatro JW cada (produzidos pelos 2 JW do nível 2, gerados a partir dos QN do nível 1), e quatro grupos com um JW (provindos dos QN do nível 2), totalizando 28 JW.

Em geral, combinando a quantidade de QN e JW para esboçar o custo dos níveis 3, 4, 5, ..., e resolvendo a recorrência, o custo obtido para um nível arbitrário i , onde $0 < i < \ell$, é dado pela

equação 3.6, onde o custo no último nível ℓ inclui a execução de um laço aninhado, com mostrado na equação 3.7.

$$QN_i(n) = 2^i \cdot P(n) + 2^i \cdot QN(n) + \left(\sum_{t=1}^i 2^{2i-t-1} \right) \cdot JW(n, m) + 2^i \cdot f(n) \quad (3.6)$$

$$QN_\ell(n) = 2^\ell \cdot P(n) + 2^\ell \cdot (QN(n) + NL(\mu^2)) + \left(\sum_{t=1}^{\ell} 2^{2\ell-t-1} \right) \cdot (JW(n, m) + NL(\mu^2)) + 2^\ell \cdot f(n) \quad (3.7)$$

De acordo com a equação 3.7, o algoritmo *QuickNearest*, como esperado para qualquer operador de junção, apresenta complexidade de tempo proporcional a $O(\mu^2)$, similar ao custo $O(n^2)$ encontrado em uma junção por laço aninhado. Contudo, o algoritmo *QuickNearest* permanece mais rápido uma vez que $\mu \ll n$, isto é, a operação quadrática executada pelo *QuickNearest* recebe um conjunto de cardinalidade muito menor (μ) que o tamanho total da entrada.

O melhor caso do algoritmo *QuickNearest* considera que, a cada chamada recursiva, ambas as partições criadas pela função `partition()` são igualmente balanceadas. Desta forma, as partições contêm, respectivamente, $\lceil n/2 \rceil$ e $\lfloor n/2 \rfloor$ elementos cada uma. Portanto, cada invocação recursiva separa os elementos do espaço de busca em duas metades iguais até que a cardinalidade de cada metade seja menor ou igual ao limiar de sublimação H . Tal divisão de dados culmina em uma árvore de recursão onde o número máximo de níveis é equivalente a $O(1 + \lceil \lg(n) - \lg(H) \rceil)$ (ou $O\left(1 + \left\lceil \lg \frac{n}{H} \right\rceil\right)$), onde, considerando que o nível zero é a raiz da árvore, o último nível será $\lceil \lg(n) - \lg(H) \rceil$.

Devido ao comportamento estocástico do algoritmo *QuickNearest* em cada chamada recursiva, a ocorrência do melhor caso torna-se extremamente rara, assim como a ocorrência do pior caso. No pior caso, as duas partições criadas em cada chamada recursiva contêm, respectivamente, um único elemento, que é o pivô, e o restante do conjunto de dados. Neste caso, os elementos são particionados $(n - H)$ vezes, o que resulta em $(n - H + 1)$ partições no total, onde $(n - H)$ partições contêm apenas o elemento pivô e uma única partição contém H elementos.

3.7 Experimentos

A fim de avaliar o desempenho da técnica proposta neste capítulo, o algoritmo *QuickNearest* foi comparado com duas outras abordagens que integram o estado da arte. Para realizar esta avaliação, foram consideradas as duas versões de particionamento do *QuickNearest*: por hiperplano generalizado, denotada por QNH, e por bolas métricas, denotada por QNB.

O primeiro método concorrente é o algoritmo padrão de junção *kNN* usando um laço aninhado (*nested-loop join* - NLJ). Embora esse método apresente custo computacional quadrático, ele é o único algoritmo da literatura (além do *QuickNearest*) capaz de processar junções *kNN* sobre

qualquer tipo de dado (multidimensional ou puramente métrico), usando qualquer função de distância (métrica ou não métrica).

O segundo método concorrente emprega uma estrutura de dados para indexar e processar o operador de junção kNN . Para fins de indexação, foi adotada a estrutura Slim-Tree [81], com páginas dispostas apenas em memória primária. Esta versão indexada da junção (IDX) foi escolhida porque, muitas vezes ela é capaz de melhorar o desempenho dos operadores de busca por similaridade, impondo apenas que a função de distância seja também uma métrica.

Com relação à existência de outros métodos voltados para consultas do tipo kNN , como visto no capítulo 2, muitos processam apenas um tipo de dado específico (vetores, cadeias de caracteres etc.), empregam funções de distância específicas (Minkowski, distância de edição), ou então se destinam a computar outras operações que não um kNN join, como seleções, *range joins* etc. Portanto, tal como o NLJ, o algoritmo *QuickNearest* foi projetado para ser uma técnica de propósito geral e independente de domínio. Isto faz com o que o algoritmo proposto não esteja restrito apenas a tipos de dados ou métricas particulares.

Os experimentos foram executados em um computador com processador Intel® Core™ i7-4770, 3.4 GHz, com 16 GB de RAM e sistema operacional GNU Linux, distribuição Ubuntu 14.04. Todos os algoritmos foram implementados em C++ usando o mesmo *framework*, de modo que eles podem ser diretamente comparados.

Os testes realizados analisam os seguintes aspectos do operador k -nearest neighbor join: escalabilidade (seção 3.7.1), dimensionalidade dos dados (seção 3.7.2), distribuição dos dados no espaço (seção 3.7.3), junção de dados puramente métricos (seção 3.7.4) e variações dos parâmetros do algoritmo *QuickNearest* (seção 3.7.5). Cada experimento é apresentado seguido da descrição de sua metodologia empregada, conjuntos de dados, resultados obtidos e discussão. Exceto quando mencionado ser diferente, todos os resultados reportados consideram como padrão uma operação de junção 10-NN ($k = 10$), a métrica L_2 para comparar elementos e o limiar de sublimação H (usado nos algoritmos 3.1 e 3.3) com valor 50.

3.7.1 Escalabilidade

O primeiro conjunto de testes tem como objetivo avaliar a escalabilidade dos algoritmos de junção kNN . Para isto, a cardinalidade do conjunto de entrada submetida ao operador de junção foi gradativamente aumentada, enquanto que a quantidade de comparações entre elementos e o tempo total de execução foram aferidos.

Nesta avaliação, foi empregado o conjunto GEONet/USGS², que é composto por 9144294 pontos bidimensionais representando valores de latitude e longitude de localizações ao redor do globo.

A figura 3.6 ilustra os resultados obtidos nos testes, representados em escala logarítmica. O primeiro conjunto de elementos contendo parte do conjunto GEONet/USGS submetido aos algoritmos avaliados tem 10000 elementos. A seguir, a cardinalidade foi linearmente aumentada em quanti-

²GEONet: <http://geonames.nga.mil/gns/html/namefiles.html>; USGS: https://geonames.usgs.gov/domestic/download_data.htm - Acesso: 14/Maio/2016

Figura 3.6: Avaliação da escalabilidade – conjunto GEONet/USGS

dades de 10000 elementos, até atingir 100000 pontos. Então, o incremento realizado passou para 100000, até chegar a 1000000 de pontos; e assim sucessivamente, até que 100% dos elementos do conjunto GEONet/USGS fossem considerados.

A figura 3.6(a) apresenta o tempo total que cada algoritmo testado consumiu para completar a operação de junção kNN . O NLJ foi a técnica mais lenta. A versão indexada, IDX, foi, em média, 4,99 vezes mais rápida que o NLJ. De fato, é possível notar como o uso de uma estrutura de indexação melhora o desempenho do algoritmo NLJ. Na maioria dos casos, este comportamento é observado porque o IDX se beneficia da propriedade da desigualdade triangular e poda regiões do espaço, reduzindo o número de comparações em relação ao NLJ, como pode ser visto na figura 3.6(b).

Ainda na figura 3.6(a), ambos os métodos NLJ e IDX foram superados pelas duas variações do algoritmo *QuickNearest*. Enquanto que o QNH foi 2 ordens de magnitude mais rápido que os métodos NLJ e IDX, a versão QNB foi de 3 a 4 ordens de magnitude mais rápida que eles. Numericamente, o QNH foi, em média, 95,81 vezes mais rápido que o NLJ e 18,29 vezes mais rápido que o IDX; enquanto que o QNB foi, em média, 1235,31 vezes mais rápido que o NLJ e 234,69 vezes mais rápido que o IDX. Como exemplo, o QNB gastou 36,06 segundos para realizar a junção kNN sobre 800000 objetos, enquanto que o NLJ levou praticamente o mesmo tempo (36,49 segundos) para operar sobre apenas 20000 objetos. Por sua vez, o IDX gastou 33 segundos para completar a operação em 40000 elementos. Considerando, agora, os 800000 objetos, os métodos NLJ, IDX e QNH gastaram, respectivamente, 18 horas, 3,4 horas e 5,5 minutos.

Analisando o tempo total necessário para completar a operação de junção em todo o conjunto de dados, o QNB foi o método mais rápido, enquanto que o método mais lento foi o NLJ. O QNB processou a junção kNN sobre cerca de 10 milhões de objetos em aproximadamente 2,2 horas, enquanto que o NLJ levou 98,03 dias para realizar o mesmo cômputo. Por sua vez, a versão indexada IDX gastou cerca 18,71 dias para retornar o mesmo resultado.

Embora ambos os métodos QNB e QNH realizem a operação exata (não-aproximada) de junção kNN , estes algoritmos são estocásticos. Visto que eles são baseados em uma estratégia de escolha aleatória para selecionar os elementos de referência e dividir o espaço de busca, alguns pivôs podem estar melhor localizados do que outros, gerando partições mais balanceadas ou menos balanceadas.

Em termos práticos, como os algoritmos são recursivamente invocados, as partições criadas em cada chamada nem sempre serão boas partições (igualmente balanceadas), mas também não serão sempre partições ruins (extremamente desbalanceadas). É possível observar o comportamento da diferença de “qualidade” das partições no gráfico da figura 3.6(a). Assim, enquanto que para os métodos NLJ e IDX as respectivas curvas são representadas por uma linha reta, pequenas oscilações estão presentes nas curvas do QNB (entre as cardinalidades de 5 e 9,1 milhões) e do QNH (entre as cardinalidades de 400000 e 3 milhões).

Comparando os algoritmos QNB e QNH, nota-se que o primeiro foi, em média, 11,87 vezes mais rápido que o segundo. Este ganho é explicado pelo fato das janelas produzidas pelo algoritmo 3.7 possuírem, de fato, mais elementos do que a quantidade necessária para processar a junção. Com isso, o algoritmo QNH realiza mais comparações (como também pode ser constatado na figura 3.6(b)) e torna-se mais lento que o QNB.

A figura 3.6(b) apresenta o número de comparações (cálculos de distância) realizado pelos vários algoritmos de junção kNN . Como o tempo de execução é diretamente influenciado pelo número de cálculos de distância, é esperado que os métodos mais lentos da figura 3.6(a) sejam aqueles que realizam mais comparações. De fato, este comportamento pode ser observado na figura 3.6(b).

O número de comparações realizado pelo QNH foi, em média, 2 e 4 ordens de magnitude menor que os totais dos algoritmos IDX e NLJ, respectivamente. De forma similar, o total de cálculos de distância realizado pelo QNB foi, em média, 1, 3 e 4 ordens de magnitude menor que os valores obtidos para os algoritmos QNH, IDX e NLJ, respectivamente.

Novamente, o QNH executou mais comparações que o QNB porque, em espaços métricos genéricos, não é possível computar as distâncias exatas entre um objeto e um hiperplano, mas é possível estimar um limite inferior para esta distância (equação 3.1). Assim, como o QNH realiza comparações desnecessárias, este valor aumenta em relação ao alcançado pelo QNB e pelo IDX. Tal comportamento pode ser observado na figura 3.6(b). Para os valores de cardinalidade de 10000 e 20000 objetos, o QNH realiza mais comparações que o IDX. Enquanto o IDX realizou 5854774 e 18793750 comparações para 10000 e 20000 objetos, respectivamente, o QNH realizou 7843105 (cerca de 34% a mais) e 19094398 (cerca de 2% a mais) comparações, para os mesmos valores de cardinalidades.

Para 10000 e 20000 objetos, o algoritmo IDX reduziu o número de comparações realizado em relação ao QNH. Entretanto, ainda assim o IDX consumiu mais tempo que o QNH para completar a operação. Isto ocorre porque poucos pares presentes nas janelas do QNH são incluídos no conjunto-resposta, enquanto que o IDX inclui um grande número de pares. Ao processar consultas do tipo kNN , a inclusão de pares no conjunto-resposta torna-se computacionalmente cara, visto que a estrutura de dados do conjunto-resposta precisa manter a ordenação dos pares a fim de realizar as substituições daqueles de menor similaridade por aqueles mais similares. Assim, para cardinalidades de 10000 e 20000 objetos, o QNH foi, respectivamente, 1,55 e 6,27 segundos mais rápido que o IDX. Além disso, o IDX se comportou melhor que o NLJ para conjuntos de dados de baixa dimensionalidade. Contudo, conforme a dimensionalidade dos dados aumenta, o IDX não consegue se beneficiar da propriedade da desigualdade triangular para podar elementos e perde desempenho, como discutido na seção 3.7.2 a seguir.

Figura 3.7: Avaliação da dimensionalidade – conjunto Nasa

3.7.2 Dimensionalidade dos dados

O conjunto de dados GEONet/USGS usado no teste de escalabilidade contém cerca de 10 milhões de pontos, o que é bastante razoável para se avaliar a escalabilidade do operador *QuickNearest*. Entretanto, este conjunto de dados é composto por valores de latitude e longitude, e estes pontos são dispostos em um espaço bidimensional. Embora este conjunto seja real, muitas aplicações que demandam por consultas por similaridade manipulam dados em dimensões maiores, e quanto mais alta a dimensionalidade do conjunto, menor é o desempenho das técnicas de consultas do tipo *kNN*. Dessa forma, o objetivo geral desta segunda bateria de experimentos é analisar o comportamento do algoritmo *QuickNearest* conforme a dimensionalidade dos dados varia.

Nesta série de experimentos foram empregados conjuntos de alta dimensionalidade e foram aferidos os mesmos aspectos considerados para o teste de escalabilidade, uma vez que o tempo total de execução é fundamental para um algoritmo de busca como o *QuickNearest* e o número de comparações é o parâmetro que o influencia mais diretamente.

Para uma análise inicial, os algoritmos de junção *kNN* foram executados usando o conjunto de dados Nasa³. Este conjunto é composto por 40150 pontos dispostos em um espaço de 20 dimensões. Cada ponto corresponde a um vetor de características de uma imagem extraída da base de dados da NASA. Os resultados obtidos são apresentados na figura 3.7.

A figura 3.7(a) apresenta o tempo total que cada algoritmo consumiu para realizar as junções. De forma similar ao teste de escalabilidade, quanto maior a cardinalidade do conjunto submetido aos algoritmos, maior o tempo necessário para finalizar o processamento. Neste teste, o algoritmo IDX apresentou o resultado mais lento. De fato, a técnica NLJ tradicional foi, em média, 4 vezes mais rápida que o IDX.

Neste ponto, é possível perceber como o aumento da dimensionalidade do conjunto afeta o desempenho do método indexado IDX. Conforme o número de dimensões cresce, o espaço de busca torna-se esparsos. Uma vez que os objetos estão dispostos ao longo de mais dimensões e tendem a ficar muito afastados uns dos outros, as distâncias entre eles crescem tanto que as condições para

³Nasa: <http://www.sisap.org/dbs.html> - Acesso: 11/Setembro/2016

Figura 3.8: Avaliação da dimensionalidade – conjunto Aloï

poda de elementos usando desigualdade triangular não são satisfeitas pelo IDX. Portanto, como o critério de poda de regiões do espaço falha, o desempenho do algoritmo se reduz drasticamente.

Ambas as variações do algoritmo *QuickNearest* foram mais rápidas que os métodos NLJ e IDX. O QNH foi, em média, 1,35 e 5,50 vezes mais rápido que o NLJ e IDX, respectivamente. Por sua vez, o QNB foi, em média, 2,61 e 10,46 vezes mais rápido que o NLJ e IDX, respectivamente.

A figura 3.7(b) mostra o número de comparações realizadas em cada técnica. Novamente, é esperado que as técnicas mais rápidas realizem um menor número de comparações entre os elementos, e este comportamento pode ser observado. Na figura 3.7(b), o NLJ realizou, em média, 24,10% menos comparações que o IDX, o que evidencia os problemas de uma estrutura de indexação ao manipular conjuntos de alta dimensionalidade. Ambas as técnicas QNB e QNH realizaram, respectivamente, 62,76% e 34,24% menos comparações que o NLJ. Comparando o QNB com o QNH, o primeiro foi o algoritmo mais rápido, executando 44,23% menos cálculos de distância que o segundo.

Para uma segunda análise do aspecto da dimensionalidade, os algoritmos de junção *kNN* foram executados usando também o conjunto Aloï⁴. Este conjunto é composto por imagens de 1000 objetos distintos, rotacionados em ângulos de 5 graus, totalizando 72000 imagens distintas. O conjunto Aloï foi representado por 144 características de momentos de cor [79]. Os resultados obtidos são mostrados na figura 3.8, onde a cardinalidade do conjunto de entrada cresce em passos de 8000 elementos.

A figura 3.8(a) apresenta os tempos de execução de cada algoritmo considerado. Neste experimento, o NLJ foi o método mais lento. O NLJ foi, em média, 1,13 vezes mais lento que a versão indexada IDX, para todos os valores de cardinalidade.

Embora fosse esperado que o desempenho do IDX diminuísse devido à alta dimensionalidade do conjunto, é importante destacar que os objetos do conjunto Aloï foram inseridos no índice de modo tendencioso para melhorar o desempenho do método IDX, fazendo com que objetos mais similares entre si fossem inseridos sequencialmente. Desta forma, o IDX foi capaz de reduzir as

⁴Aloï: <http://aloi.science.uva.nl/> - Acesso: 10/Setembro/2016

Figura 3.9: Avaliação da dimensionalidade – conjunto Faces

regiões de sobreposição do espaço métrico e melhorar o desempenho da estrutura de indexação em relação ao que ocorreria se os dados fossem inseridos de forma aleatória. Na prática, contudo, os elementos de uma base de dados são geralmente inseridos em ordem aleatória, e o ganho de desempenho visto na inserção tendenciosa não é observado.

Ainda na figura 3.8(a), ambos os algoritmos *QuickNearest* superaram os métodos NLJ e IDX. O QNH foi, em média, 1,50 e 1,32 vezes mais rápido que o NLJ e IDX, respectivamente; enquanto que o QNB foi, em média, 3,25 e 2,86 vezes mais rápido que o NLJ e IDX, respectivamente. Para o algoritmo QNB, este ganho corresponde a 1 ordem de magnitude nos valores de cardinalidade superiores a 70000, como mostrado na figura 3.8(a).

A figura 3.8(b) mostra o número de comparações realizadas por cada método ao processar o conjunto Aloï. O algoritmo IDX executou 31,89% menos cálculos de distância que o NLJ, em média. Novamente, ambas as variações do *QuickNearest* superaram os NLJ e IDX. O QNH executou, em média, 34,66% e 4,02% menos comparações que o NLJ e IDX, respectivamente. Já o QNB executou, em média, 69,68% e 55,43% menos comparações que o NLJ e IDX, respectivamente.

Nos testes da figura 3.8 é possível observar que a capacidade de poda da estrutura de indexação leva a uma menor quantidade de comparações do IDX sobre o NLJ e a um tempo de execução menor. Embora na figura 3.8(b) a diferença entre os algoritmos QNH e IDX seja de cerca 4%, também é possível notar que o primeiro algoritmo manteve-se consistentemente mais eficiente que o segundo.

Em uma análise final da dimensionalidade, os algoritmos de junção *kNN* foram executados sobre o conjunto de dados Faces⁵. Este conjunto é composto por 1016 vetores de características, cada um possuindo 761 dimensões. Antes de serem processados pelos algoritmos de junção, os vetores foram previamente embaralhados de modo aleatório para evitar o comportamento da inserção tendenciosa reportada para o conjunto Aloï. Os resultados obtidos são apresentados na figura 3.9.

Analisando a figura 3.9(a), que mostra o tempo de execução total dos algoritmos, é possível notar o mesmo comportamento encontrado na figura 3.7(a), onde o IDX foi a técnica mais lenta e ambas

⁵Faces: <http://www.sisap.org/dbs.html> - Acesso: 12/Setembro/2016

as variantes do *QuickNearest* foram novamente as mais rápidas. Este comportamento reforça o argumento principal de que quanto mais alta a dimensionalidade do conjunto de dados, pior é o desempenho da estrutura de índice do IDX. De fato, mesmo manipulando apenas 1016 elementos, o IDX foi, em média, 0,33 vezes mais lento que o NLJ. Por sua vez, o QNH foi, em média, 1,19 e 1,54 vezes mais rápido que o NLJ e IDX, respectivamente, enquanto que o QNB foi, em média, 1,45 e 1,87 vezes mais rápido que os algoritmos correspondentes.

A figura 3.9(b) mostra o número de comparações realizadas por cada algoritmo. Novamente, uma vez que o índice não é capaz de aplicar a propriedade da desigualdade triangular para realizar podas, o IDX foi a técnica que exigiu maior esforço computacional. Como exemplo, o NLJ realizou 13,44% menos comparações que o IDX. Com relação às variações do *QuickNearest*, o QNH executou, em média, 15,37% e 26,59% comparações a menos que o NLJ e IDX, respectivamente. Já o QNB realizou, em média, 31,37% e 40,56% cálculos de distância a menos que o NLJ e IDX, respectivamente.

De forma geral, a análise da dimensionalidade dos dados permite pontuar que, conforme o número de dimensões aumenta, o desempenho dos algoritmos de junção *kNN* tende a se reduzir. Para todos os algoritmos analisados, quanto maior o número de dimensões, mais caro é o processamento da função de distância. Para casos específicos, tal como o IDX, a estrutura de indexação responsável por manipular os dados degenera-se e leva a uma redução de desempenho. Ainda assim, ambos os algoritmos *QuickNearest* propostos alcançaram os melhores valores de desempenho para todos os conjuntos de dados considerados e para todas as variações de dimensionalidade.

3.7.3 Distribuição de dados

Esta seção tem como objetivo avaliar se a maneira em que os dados estão dispostos no espaço de busca, i.e., sua distribuição, influencia o algoritmo *QuickNearest*.

Nesta avaliação, os algoritmos de junção *kNN* foram executados sobre dois conjuntos de dados sintéticos com 100000 pontos bidimensionais. O primeiro conjunto contém pontos gerados seguindo a distribuição Uniforme e o segundo é composto por pontos que seguem a distribuição Normal. Os mesmos aspectos considerados nos experimentos anteriores foram aferidos: tempo e quantidade de comparações entre pares de elementos. Os resultados desta análise de distribuição de dados são mostrados na figura 3.10.

A figura 3.10(a) apresenta o tempo total consumido por cada técnica para completar a operação de junção *kNN*. Analisando os resultados da distribuição Uniforme, observa-se que o QNB foi a técnica mais rápida. De fato, ele foi 1,20, 62,45 e 272,83 vezes mais rápido que o QNH, IDX e NLJ, respectivamente. Para o conjunto seguindo a distribuição Normal, o QNB permaneceu como o método mais eficiente, sendo 4,36, 65,71 e 253,50 vezes mais rápido que o QNH, IDX e NLJ, respectivamente.

Comparando os métodos da literatura na figura 3.10(a), o IDX foi 4,37 vezes mais rápido que o NLJ, o que ressalta o aspecto de que para conjuntos em baixas dimensões, como os conjuntos sintéticos

Figura 3.10: Avaliação da distribuição de dados – conjuntos sintéticos

Uniforme e Normal, a estrutura de indexação se comporta de modo satisfatório, embora ambos os algoritmos NLJ e IDX tenham sido mais lentos que as duas variantes do algoritmo *QuickNearest*.

Ainda na figura 3.10(a), o tempo de execução obtido pelos algoritmos QNB, IDX e NLJ para ambas as distribuições Normal e Uniforme foram praticamente idênticos. De fato, a maior variação entre eles é menor que 1%. Entretanto, para o QNH, o conjunto de dados com distribuição Uniforme foi processado 3,89 vezes mais rápido que o conjunto seguindo distribuição Normal. Este comportamento é explicado pela concentração de um grande volume de dados próximos aos hiperplanos gerados pelo algoritmo. Desta forma, o algoritmo 3.7 seleciona uma grande quantidade de elementos para computar a operação de junção, e um número elevado de falsos positivos é descartado. Embora este comportamento contribua para reduzir o desempenho do algoritmo QNH, ele ainda foi mais rápido que os dois concorrentes (IDX e NLJ) em 1 ordem de magnitude.

A figura 3.10(b) apresenta o número de cálculos de distância realizado pelos algoritmos testados. É possível observar que os benefícios da estrutura de indexação reduziu em 21,15 vezes o número de comparações realizadas pelo IDX em relação ao NLJ, para ambas as distribuições. Além disso, o IDX realizou 5% menos comparações ao manipular os dados em distribuição Uniforme. A concentração de pontos da distribuição Normal fez com que o índice se degenerasse mais do que na distribuição Uniforme, uma vez os dados não foram inseridos de modo agrupado. Assim, o índice possui mais regiões de sobreposição para os dados da distribuição Normal do que no conjunto Uniforme.

A concentração dos pontos na distribuição Normal também afetou o algoritmo QNH, elevando em 25,88% o número de comparações em relação à distribuição Uniforme. Entretanto, esta quantidade de comparações ainda é significativamente menor que o total dispendido pelos algoritmos IDX e NLJ. Por sua vez, o algoritmo QNB mostrou não ser afetado pela distribuição dos dados. Além disso, o QNB não apresentou quedas de desempenho devido à concentração de pontos da distribuição Normal, visto que o algoritmo 3.10 seleciona apenas os elementos que certamente devem ser analisados, evitando processar falsos candidatos.

Figura 3.11: Avaliação de dados puramente métricos – conjunto UniProt

3.7.4 Dados adimensionais

Os testes anteriores analisaram o algoritmo *QuickNearest* ao manipular conjuntos de alta dimensionalidade e com diferentes distribuições. Contudo, é comum que consultas por similaridade sejam executadas em conjuntos de dados que não possuem dimensão (adimensionais) e nem distribuição. Estes dados também são chamados de *dados puramente métricos* [93]. Geralmente, dados puramente métricos não são representados por vetores de características, não podem ser dispostos em espaços n -dimensionais e são comparados por funções de distâncias distintas das funções L_p . Desse modo, os testes apresentados nesta seção investigam o comportamento do algoritmo *QuickNearest* ao processar dados puramente métricos.

Para esta análise, foi empregada uma parte do conjunto de dados UniProt⁶, onde 17500 proteínas distintas foram selecionadas aleatoriamente. As sequências proteicas são representadas por letras do alfabeto latino, e a distância de edição (capítulo 2) foi adotada como métrica para computar o valor de dissimilaridade entre os elementos. Como a distância de edição é computacionalmente cara de ser calculada (quadrática), é esperado que os resultados apresentem um desempenho inferior aos alcançados pelos outros experimentos que usaram as distâncias L_p (linear). Os resultados obtidos são apresentados pela figura 3.11.

A figura 3.11(a) apresenta o tempo total de cada método ao processar a junção kNN . Conforme a cardinalidade do conjunto aumenta, o desempenho de todos os algoritmos se reduz. Neste experimento, o IDX foi a técnica mais lenta. A estrutura de índice se degenera rapidamente e perde desempenho. Isto acontece porque, uma vez que a distância de edição retorna valores discretos, vários pares de proteínas possuem o mesmo valor de distância, o que faz com que todos precisem ser verificados. Como o cálculo da função de distância é caro, assim que o algoritmo IDX ultrapassou o tempo consumido pelo NLJ, sua execução foi interrompida. A justificativa é que, como o algoritmo NLJ realiza a comparação entre todos os pares de objetos, i.e., um produto Cartesiano, qualquer outro método que seja mais lento que o NLJ não é factível para ser aplicado; explicando a ausência do IDX nos resultados da figura 3.11.

⁶UniProtKB: http://www.uniprot.org/uniprot/?query=*&fil=reviewed:yes - Acesso: 13/Junho/2016

Ainda analisando a figura 3.11(a), vê-se que ambas as variações do *QuickNearest* foram mais rápidas que o NLJ (e IDX). O QNH foi 1,42 vezes mais rápido que o NLJ, enquanto que o QNB foi 1,73 vezes mais rápido. Comparando o QNB com o QNH, o primeiro foi 1,24 vezes mais rápido que o segundo.

A figura 3.11(b) apresenta o número de comparações realizadas no cômputo da junção. Novamente, a execução do algoritmo IDX não foi finalizada e, por isso, nenhum valor foi medido para esta técnica, mas é possível afirmar que tal valor é superior àquele alcançado pelo NLJ. Como reflexo do maior consumo de tempo, o algoritmo NLJ também executou o maior número de comparações que ambas as variações do *QuickNearest*. O QNH realizou, em média, 35,97% menos comparações que o NLJ. Por sua vez, o QNB realizou, em média, 44,09% menos cálculos de distância que o NLJ. Quando diretamente comparados, o QNB realizou 11,94% menos cálculos que o QNH.

3.7.5 Variação paramétrica

Uma vez que o *QuickNearest* é baseado na estratégia de divisão e conquista, esta seção tem como objetivo avaliar o único parâmetro capaz de influenciar o desempenho do algoritmo: o limiar de sublimação H , presente nos algoritmos 3.1 e 3.3. Foram considerados valores distintos para o parâmetro de consulta k e para cada valor de k variou-se também o parâmetro H . O algoritmo *QuickNearest* foi executado para cada combinação de valores de k e H , medindo o tempo total de processamento e o número de comparações realizadas.

Para esta avaliação, foram selecionados aleatoriamente 100000 pontos do conjunto GEONet/USGS (descrito na seção 3.7.1). As figuras 3.12 e 3.13 mostram os resultados obtidos. Os resultados estão apresentados de modo que a primeira coluna de gráficos ilustra o tempo de execução do algoritmo QNB, a segunda coluna traz o tempo de processamento do QNH e a terceira coluna sumariza o número total de cálculos de distância destes algoritmos. De forma análoga, cada linha de gráficos corresponde a um valor distinto do parâmetro k .

Nas figuras 3.12(a), 3.12(b) e 3.12(c), o algoritmo *QuickNearest* foi executado para o valor de $k = 1$. Para o QNB, o melhor valor de desempenho foi alcançado para $H = 20$. Este valor indica que os algoritmos 3.1 e 3.3 realizam o particionamento dos 100000 elementos do conjunto em subconjuntos menores enquanto a cardinalidade de cada conjunto individual for maior que 20. Então, quando a cardinalidade for menor ou igual a este valor, a junção é de fato finalizada pela invocação de um laço aninhado. De modo similar, o algoritmo QNH atingiu o melhor valor de tempo para o limiar de sublimação $H = 80$. Como método mais rápido, o QNB foi, em média, 5,47 vezes mais rápido que o QNH. De fato, o primeiro realizou 70,98% menos comparações que o segundo.

Ambas as variações do algoritmo *QuickNearest* se comportaram de forma similar para o valor de $k = 2$, como pode ser observado pelas figuras 3.12(d), 3.12(e) e 3.12(f). Os valores do limiar de sublimação que resultaram no melhor desempenho foram, respectivamente, 30 e 90 para o QNB e QNH.

Nas figuras 3.12(a) a 3.12(f), o desempenho de ambos os algoritmos caiu quando H assumiu valores pequenos. Por exemplo, o QNB apresentou uma execução mais lenta para valores de

Figura 3.12: Variação de parâmetros: k e H .

$H < 10$. A principal razão deste comportamento é que partições pequenas (10 elementos ou menos), quando novamente divididas, sofrem de desbalanceamento mais severo, apresentando cardinalidade menor que k . Nestes casos, o laço dos passos 5-9 do algoritmo 3.1 é executado diversas vezes, reduzindo o desempenho. Para o QNH, o desempenho cai para valores de $H < 50$. De fato, o QNH necessita de mais elementos para que seja possível selecionar pivôs adequados e produzir partições com ao menos k elementos em cada uma. Se a cardinalidade das partições forem menores que este valor, o algoritmo 3.1 executa o laço dos passos 5-9 mais de uma vez. Além disso, as janelas produzidas pelo algoritmo 3.7 contêm mais elementos falsos positivos,

Figura 3.13: Variação de parâmetros: k e H (continuação da figura 3.12).

devido à inexactidão da distância de um objeto ao hiperplano. Estes mesmos comportamentos dos algoritmos QNB e QNH também são observados nas figuras 3.12(g) a 3.12(l).

A figura 3.13 apresenta a avaliação do algoritmo *QuickNearest* para valores de k variando entre 15 e 30. Em consultas do tipo kNN , é incomum que o parâmetro k assuma valores grandes (e.g. maiores que 30), pois isto gera um conjunto resposta com muitos elementos e o usuário pode ser levado à reformulações de consultas [70] frequentes.

Em todos os experimentos apresentados na figura 3.13, o limiar H recebeu valores superiores a $2 \cdot k$. A sugestão é que este parâmetro assuma valores superiores a $3 \cdot k$ para que o algoritmo 3.1 tenha uma quantidade de elementos suficientemente grande para particionar o espaço de busca e não proceder com execuções sucessivas do laço dos passos 5-9.

Na figura 3.13 é possível observar também que o algoritmo QNB executa mais rapidamente para valores pequenos de H , enquanto o algoritmo QNH mostra-se mais rápido para valores de H no intervalo $[70, 100]$. Conforme H aumenta, ambos os algoritmos 3.1 e 3.3 realizam a invocação do laço aninhado mais rapidamente, ou seja, com menos divisões do espaço, uma vez que a cardinalidade das partições pode ser maior. Assim, ambos QNB e QNH vão tornando-se computacionalmente mais caros e o desempenho de ambos tende a ser o mesmo do NLJ. Na realidade, para um valor de H maior que a cardinalidade do conjunto de dados, o algoritmo *QuickNearest* executa a operação de laço aninhado diretamente, sem realizar particionamentos do espaço. Entretanto, ainda assim, seu desempenho nunca é inferior ao do algoritmo NLJ.

3.8 Considerações finais

Este capítulo apresentou o *QuickNearest*, um algoritmo eficiente para computar a operação de junção por similaridade do tipo kNN .

Ao contrário dos métodos tradicionais para processamento de *k-nearest neighbor joins*, o algoritmo *QuickNearest* é independente do domínio dos dados, ou seja, é capaz de processar qualquer tipo de dados, usando qualquer função de distância, tanto métrica quanto aquelas que atendem apenas às propriedades de reflexividade, simetria e não negatividade.

A implementação base do algoritmo *QuickNearest* é composta por duas funções principais (`quickNearest()` e `joinWindows()`). Ambas as funções realizam ou o particionamento do espaço (através da função `partition()`) ou a operação de laço aninhado. Foram apresentadas duas estratégias para particionamento dos elementos do espaço de busca: a divisão por hiperplano generalizado e o particionamento por bolas métricas. Ambas as estratégias possuem sua própria implementação das funções `windowDistance()` e `computeWindows()`, responsáveis pela criação das janelas de objetos de junção.

O algoritmo *QuickNearest* foi avaliado sob diversos aspectos. A análise de escalabilidade mostrou que a técnica proposta chega a ser de 3 a 4 ordens de magnitude mais rápida que os métodos encontrados na literatura. A análise de dimensionalidade mostrou que o algoritmo se mantém superior às técnicas concorrentes mesmo para tratar dados em dimensionalidade elevada. Na avaliação da distribuição do conjunto de dados, o *QuickNearest* mostrou-se invariante a este fator e permaneceu sendo a técnica mais eficiente. Por fim, o *QuickNearest* também foi capaz de processar dados puramente métricos com eficiência. Em uma análise final, foi verificado como o limiar de sublimação H , que controla o particionamento dos dados, impacta no desempenho do *QuickNearest* geral.

Apoiado nos resultados obtidos, é possível apontar que o algoritmo *QuickNearest* é a técnica mais eficiente para processamento de junções kNN . Ainda assim, um problema associado aos operadores

de junção por similaridade é o grande volume de dados retornado. Para contornar essa questão, o capítulo 4 seguinte propõe um operador binário que, através da exploração de conceitos de ordenação internamente a um operador por similaridade, permite controlar a cardinalidade do resultado final, levando a conjuntos-resposta mais intuitivos, melhor ajustado às necessidades de aplicações de análise de dados e que, para essas aplicações, obtêm resultados de melhor qualidade.

Wide-join

*“Com ordem e tempo,
descobre-se o segredo de
tudo fazer, e fazer bem.”*

Pitágoras

Este capítulo apresenta a segunda contribuição desta tese, um conjunto de operadores binários que estendem os operadores de junção por similaridade, chamados de *wide-joins*. Um *wide-join* é um operador binário que estende o conceito de junção através da exploração de operações colocadas intermediárias à operação de produto Cartesiano e à operação de seleção que define formalmente a junção, incluindo por exemplo um operador de ordenação. Os operadores de junção por similaridade normalmente devolvem conjuntos-resposta de alta cardinalidade que, com grande frequência, são posteriormente filtrados em operações subsequentes na expressão de uma consulta. Ou seja, junções por similaridade são aplicadas principalmente como operadores de préprocessamento, seguidas de filtrações adicionais sobre o seu resultado. O operador *wide-join* aborda esse problema incorporando condições de filtração internamente para o controle da cardinalidade do conjunto-resposta e da seleção dos pares de elementos mais similares em contexto geral.

A seção 4.1 delinea as bases do operador *wide-join*, apresentando os fundamentos de sua formulação. Na seção 4.2 é apresentada a definição formal do operador *wide-join*. Como operador de consulta, o *wide-join* realiza a comparação de atributos de objetos através de predicados. Os tipos de predicados que podem compor o *wide-join* são discutidos na seção 4.3. A seção 4.4 realiza a avaliação experimental dos operadores *wide-join* e seus predicados. A seguir, na seção 4.5, é apresentada uma extensão do operador *wide-join* para tratar junções de um conjunto consigo próprio, o operador *self wide-join*, sendo que esta versão foi utilizada na formulação de um modelo de *software* para detecção de quase-duplicatas. Algumas propriedades algébricas relacionadas à otimização do operador *wide-join* são mostradas na seção 4.6. Por fim, a seção 4.7 apresenta as considerações finais do capítulo.

4.1 Derivação do operador *wide-join*

A partir da observação dos três tipos de junção por similaridade definidos na seção 2.4, pode-se perceber que os operadores *range join* e *k-nearest neighbor join* empregam os mesmos operadores de comparação por similaridade usados pelos operadores *range selection* e *k-nearest neighbor selection*; respectivamente, os operadores de comparação *similarity range* (Rng) e *k-nearest neighbor* (kNN). Já o operador *k-closest neighbor join* não tem correspondência com um operador “*k-closest neighbor selection*”, como foi apontado na tabela 2.4. Isso pode ser resolvido de duas maneiras.

A primeira abordagem considera que existe um operador de comparação “*k-closest neighbor*”: $kCN(d, k)$. Nesse caso, dados uma relação R , um atributo complexo $S \subseteq R$ amostrado num conjunto de dados complexos $\mathcal{S}$, e um elemento de consulta s_q , um operador de seleção que faça uso do operador de comparação kCN produziria um “*k-closest neighbor selection*”, $\sigma_{(SkCN(d,k)s_q)}R = \{t_1, t_2, \dots, t_k\}$, tal que:

- $t_1 = \{t \in R \mid \forall r \in R : d(s_q, t.S) < d(s_q, r.S)\}$
- $t_2 = \{t \in R - \{t_1\} \mid \forall r \in R - \{t_1\} : d(s_q, t.S) < d(s_q, r.S)\}$
- $t_3 = \{t \in R - \{t_1, t_2\} \mid \forall r \in R - \{t_1, t_2\} : d(s_q, t.S) < d(s_q, r.S)\}$
- $t_4 = \{t \in R - \{t_1, t_2, t_3\} \mid \forall r \in R - \{t_1, t_2, t_3\} : d(s_q, t.S) < d(s_q, r.S)\}$
- ...
- $t_k = \left\{ t \in R - \left\{ \bigcup_{i=1}^{k-1} t_i \right\} \mid \forall r \in R - \left\{ \bigcup_{i=1}^{k-1} t_i \right\} : d(s_q, t.S) < d(s_q, r.S) \right\}$

Esta primeira solução é consistente do ponto de vista conceitual, pois pode-se verificar que $\sigma_{(SkCNs_q)}R \subseteq \sigma_{(SkNNS_q)}R$ (basta comparar os desmembramentos acima com os da definição 8) e, portanto, ambos os operadores de comparação kCN e kNN produziriam o mesmo resultado quando usados para compor um operador de seleção, sendo necessário a existência de apenas um deles. Eles somente geram resultados diferentes quando existe mais de um centro de busca.

A segunda solução é considerar que o operador tratado na literatura como *k-closest neighbor join* não é de fato um operador de junção. Isso pode ser percebido analisando a definição conceitual do operador de junção. Como enunciado pela definição 9, um operador de junção é definido como o produto Cartesiano das duas relações envolvidas seguido de uma operação de seleção com o predicado da junção. De fato, os operadores *range join* e *k-nearest neighbor join* podem ser definidos assim. Mas a definição do operador *k-closest neighbor join* demanda uma seleção adicional: ele requer que os pares candidatos a compor a resposta sejam computados (produto Cartesiano) e, sobre este primeiro resultado é necessário ordenar as tuplas pela distância entre os elementos originais que compõem cada par ordenado e só então aplicar uma seleção sobre as distâncias, permitindo recuperar os pares mais próximos num contexto geral.

Nota-se que a seleção executada no *k-closest neighbor join* é aplicada sobre um atributo que não faz parte das relações originais (o valor ordinal da tupla numa ordenação), o qual só pode ser calculado

depois que todo o resultado do produto Cartesiano original já houver sido obtido. Portanto, de maneira distinta do predicado que cria uma junção, que pode ser executado junto ou posterior ao produto Cartesiano, o predicado da referida seleção somente pode ser executado posteriormente à finalização do pareamento de todas as tuplas. Dessa forma, os operadores *range join* e *k-nearest neighbor join* são, de fato, operadores de junção, mas o operador *k-closest neighbor "join"* não é.

É importante destacar que o operador *k-closest neighbor join* é um operador válido, com significado prático e aplicabilidade em diversos contextos, como visto nos capítulos 1 e 2. De fato, esse operador tem aplicabilidade até maior que os outros operadores de junção por similaridade, pois, em geral, resulta em respostas com cardinalidade bem menores e de maior interesse ao usuário do que os demais operadores de junção por similaridade, como, novamente, observado nos capítulos 1 e 2. Entretanto, ele não é um operador de junção *in stricto sensu* segundo a definição de “junção” do Modelo Relacional. Apesar disso, este texto continuará a se referir a ele como junção (“join”), pois essa é a nomenclatura pela qual esse operador é conhecido na literatura.

A segunda solução é mais interessante para esta tese. Ela caracteriza a necessidade de haver mais operadores de busca para atender as consultas por similaridade, e ao mesmo tempo mostra como eles podem/devem ser desenvolvidos. O operador *k-closest neighbor join*, além de ter maior serventia que as outras junções, também pode ser descrito em álgebra relacional, o que permite manter o desenvolvimento de ferramentas que disponibilizam consultas por similaridade ainda apoiadas no modelo relacional. Assim, o *k-closest neighbor join* é expresso em álgebra relacional de acordo com a equação 4.1.

$$R \stackrel{(A_1 \text{ kCN}(d,k) A_2)}{\bowtie} S \Leftrightarrow \sigma_{(ord \leq k)} \left(\Pi_{\{R,S,\mathcal{F}(dist) \rightarrow ord\}} \left(\Pi_{\{R,S,d(A_1,A_2) \rightarrow dist\}} (R \times S) \right) \right) \quad (4.1)$$

Na equação 4.1, $d(A_1, A_2)$ é a função de distância aplicada aos atributos complexos A_1 e A_2 das relações R e S , e $\mathcal{F}$ é uma função que atribui um número sequencial ao atributo *ord* de cada tupla na relação ordenada segundo o atributo *dist*.

4.2 Definição do operador *wide-join*

Além de fornecer a definição algébrica do operador *k-closest neighbor join*, a equação 4.1 traz a base conceitual para criar todo um novo conjunto de operadores binários para busca por similaridade, que inclui a exploração do conceito de ordenação como um passo intermediário do processamento. Isso pode ser feito interpondo um operador de seleção por similaridade entre o produto Cartesiano e a primeira operação de projeção na equação 4.1, a fim de englobar junções por similaridade. Como resultado, obtém-se um novo operador binário por similaridade composto por operações adicionais além do produto Cartesiano seguido de seleção. Por executar mais operações do que a definição formal original da junção, este operador é chamado de “*Wide-join*”. Ele é formalizado pela definição 15:

Definição 15 (Wide-join: $\bowtie$). *Seja $\mathcal{S}$ um universo de objetos complexos, R e S duas relações, $A_1 \subseteq R$ e $A_2 \subseteq S$ dois atributos complexos no domínio $\mathcal{S}$, $d : \mathcal{S} \times \mathcal{S} \mapsto \mathbb{R}_+$ uma função de distância sobre $\mathcal{S}$ e $\kappa_f \in \mathbb{N}^*$*

uma constante não-nula. Um *wide-join* $R \bowtie_{(A_1 \theta_s A_2), \kappa_f} S$ é um operador binário que computa uma junção por similaridade usando o predicado de comparação por similaridade $A_1 \theta_s A_2$, ordena as tuplas intermediárias pelo valor da distância entre os atributos A_1 e A_2 e retorna as κ_f tuplas contendo A_1 e A_2 mais similares em geral. O *wide-join* é expresso em álgebra relacional de acordo com a equação 4.2:

$$R \bowtie_{(A_1 \theta_s A_2), \kappa_f} S \equiv \sigma_{(ord \leq \kappa_f)} \left(\Pi_{\{R, S, \mathcal{F}(d(A_1, A_2)) \rightarrow ord\}} \left(\sigma_{(A_1 \theta_s A_2)} (R \times S) \right) \right) \quad (4.2)$$

Na equação 4.2, $\mathcal{F}$ é a função que projeta a classificação ordinal de cada valor projetado pela função de distância d no atributo estendido ord . Nesse caso, $\mathcal{F}$ funciona como uma função que recebe como parâmetro os valores de distância entre A_1 e A_2 e retorna sua numeração ordinal. Embora o operador *wide-join*, e também o *k-closest neighbor join*, necessitem do conceito de ordem, a ordenação é realizada internamente ao processamento e de modo compatível com a teoria relacional, visto que o operador Π recebe duas relações que não consideram a ordem entre as tuplas e também devolve uma relação que não considera a ordem entre as tuplas. De fato, $\mathcal{F}$ segue as mesmas definição e modo de processamento que permitem que funções de agregação como *sum*, *avg*, *min*, *max* etc. sejam empregadas no modelo relacional. É importante destacar que as tuplas não são ordenadas fisicamente, nem em memória primária nem secundária, uma vez que o conceito de ordem fica embutido no atributo ord , que é projetado no final da operação mas permite filtrar os κ_f pares mais similares em geral.

O operador *wide-join* é normalmente processado por laços aninhados, i.e., realizando $O(|R| \cdot |S|)$ comparações para obter o resultado final. Entretanto, algumas otimizações de implementação podem ser realizadas para acelerar o processamento.

A indexação dos elementos da segunda relação de junção é uma técnica efetiva e frequentemente empregada pela literatura, podendo ser aplicada a *wide-joins*. Visto que as estruturas métricas de indexação se beneficiam de propriedades como a desigualdade triangular para podar regiões do espaço, elas permitem reduzir, entre outros, o número de operações de I/O (entrada/saída) no processamento.

A fim de reduzir o tempo de CPU durante o processamento da junção interna, também é possível empregar uma versão estendida de produto Cartesiano que devolva uma tupla $\langle r, s, d(r.A_1, s.A_2) \rangle$, ou seja, o par ordenado original mais a distância entre os elementos complexos do respectivo par já calculada. Assim, o valor de dissimilaridade entre os atributos complexos das tuplas r e s não necessita de ser recalculado nas operações posteriores, como, por exemplo, na função $\mathcal{F}$ da equação 4.2.

Por fim, as operações representadas pela projeção e seleção final no segundo membro da equação 4.2 podem ser obtidas diretamente se o resultado da junção por similaridade interna for armazenado na estrutura de dados de fila de prioridade, sendo que o valor da dissimilaridade controla a prioridade dos elementos. Dessa forma, quanto menor o valor $d(r.A_1, s.A_2)$, maior a prioridade do par. Assim, o resultado final é obtido pela remoção dos κ_f primeiros elementos da fila de prioridade.

O operador *wide-join* difere dos outros operadores de junção por similaridade por flexibilizar o controle da cardinalidade do conjunto-resposta e a expressão do predicado θ_s a ser avaliado pela junção por similaridade interna, onde cada tipo de predicado produz uma variação do operador *wide-join*, como discutido na próxima seção 4.3.

4.3 Predicados do *wide-join*

Os predicados utilizados para compor a operação por similaridade do operador *wide-join* podem ser divididos em predicados de termo único (seção 4.3.1) e predicados de termos múltiplos (seção 4.3.3). Os termos de ambos os tipos de predicados podem ainda ser negados pelo operador unário de negação, como descrito na seção 4.3.2.

4.3.1 Predicados de termo único

O tipo mais básico de *wide-join* não emprega um predicado por similaridade na junção interna, mas assume que $(A_1 \theta_s A_2)$ é sempre TRUE, $\forall r.A_1, s.A_2$. Neste caso, o operador de junção por similaridade interno à equação 4.2 torna-se um produto Cartesiano, de modo análogo ao da equação 4.1. Portanto, um *wide-join* que faz uso de um predicado TRUE resulta no operador *k-closest neighbor join*.

Os demais tipos de *wide-join* são obtidos a partir dos operadores de comparação por similaridade *similarity range* e *k-nearest neighbor*. Quando $\theta_s = (A_1 \text{Rng}(d, \xi) A_2)$, o operador resultante é o *range wide-join*. Um *range wide-join* permite combinar ambos os operadores *range join* e *k-closest neighbor join* em um operador único. Este operador único permite recuperar uma quantidade máxima de κ_f pares $\langle r, s \rangle$ tal que $d(r.A_1, s.A_2) \leq \xi$. Observa-se que tanto o operador *range join* como o operador *k-closest neighbor join*, quando isoladamente processados, não produziriam o mesmo resultado. Logicamente, o primeiro falha em limitar a cardinalidade do conjunto-resposta, enquanto que o segundo pode selecionar pares de elementos cujo valor de dissimilaridade excede o limiar ξ . Além disso, é computacionalmente mais lento processar uma composição de um *range join* seguido de uma seleção dos pares mais similares, como no segundo membro da equação 4.2, do que embutir estas operações em um operador único, como discutido na seção 4.4. A figura 4.1(a) ilustra, em um espaço de busca bidimensional, o resultado de um *range wide-join*. Na figura 4.1(a), pode-se notar que, embora o predicado do tipo *range* selecione mais elementos, representados pelos círculos, apenas os κ_f pares mais similares, representados por estrelas e losangos, são retornados.

Um predicado $\theta_s = (A_1 \text{kNN}(d, k) A_2)$ produz um *k-nearest neighbor wide-join*. De modo análogo ao *range wide join*, um *k-nearest neighbor wide-join* reduz a cardinalidade de um *k-nearest neighbor join* de $k \cdot |R|$ para κ_f . Este operador, entretanto, necessita da definição de dois parâmetros relacionados a quantidades: k , para os k vizinhos mais próximos na junção; e κ_f , para o número final de pares.

Valores de $\kappa_f \leq k$ filtram um subconjunto da resposta de um *k-nearest neighbor join*. Como o operador *wide-join* recupera os pares mais similares num contexto geral, este resultado intermediário pode vir a coincidir com a resposta obtida por um *k-closest neighbor join*, tal qual para $\theta_s = \text{TRUE}$. Se o

valor de k garante uma alta seletividade na junção interna, esta configuração permite otimizar a função $\mathcal{F}$, visto que uma quantidade reduzida de pares é selecionada.

Em espaços de distâncias (ou espaços métricos) contendo regiões densas, i.e., com alta concentração de elementos, a escolha de valores $\kappa_f \leq k$ pode produzir respostas com diversos pares de elementos muito similares entre si, levando a resultados com pouca informação agregada. Contudo, quando $\kappa_f > k$, o operador *k-nearest neighbor wide-join* assume um comportamento mais exploratório e retorna não só um subconjunto dos pares mais similares de um *k-nearest neighbor join*, mas ainda pares distribuídos ao longo de várias regiões do espaço, mesmo na presença de regiões densas. Este comportamento pode ser observado na figura 4.1(d), onde, embora haja mais candidatos possíveis da segunda relação (representados por círculos) a serem pareados com os elementos da primeira (estrelas), o operador seleciona os pares mais próximos de regiões distintas do espaço.

Embora as consultas por similaridade que fazem uso dos comparadores *similarity range* e *k-nearest neighbor* sejam as mais frequentes, os complementos destas condições são suportados pela lógica de predicados e também podem ser usados na recuperação da informação por similaridade, o que culmina nos operadores de comparação “*not-in-range*” e “*not-the-k-nearest*”, como discutido na próxima seção 4.3.2.

4.3.2 Negação de predicados de termo único

A negação de termos de um predicado existe naturalmente na álgebra relacional e é bem explorada pelas implementações de SGBDRs. Como exemplo, consultas empregando operadores como $\neq$, $\geq$ ou $>$ são comuns, sendo que é importante lembrar que o operador de comparação $\neq$ corresponde à negação de um $=$, o operador $\geq$ é a negação de um $<$, e um $>$ é a negação de um $\leq$.

Para consultas por similaridade, a negação dos operadores de comparação torna-se problemática em dois aspectos. O primeiro é que a recuperação de elementos tendo a similaridade como critério corresponde a um predicado “direto”, não a uma negação. Por exemplo, o que seria a negação de um predicado que recupera os 10 vizinhos mais próximos (10-NN)? Nota-se que recuperar os 10 *elementos mais distantes* é diferente de recuperar os *elementos que não são os 10 mais próximos*.

Como segundo problema, a negação de predicados por similaridade geralmente retorna um conjunto-resposta de altíssima cardinalidade. Como exemplo, em uma tentativa de negar um operador de comparação 10-NN, uma consulta que recupera os *elementos que não são os 10 mais próximos* tem como resposta todos os elementos presentes na base de dados, exceto pelos 10 elementos mais próximos (10-NN) ao centro de consulta. Considerando os operadores de junção por similaridade, a negação de seus predicados recupera ainda mais pares, aumentando ainda mais a cardinalidade do resultado.

Apesar destas observações, a negação de um predicado possui uma motivação interessante para consultas por similaridade: quando os elementos mais similares já são conhecidos, mas o usuário deseja *obter conhecimento adicional* sobre o conjunto de dados, *como recuperar os elementos que estão além daqueles já conhecidos*? Nota-se que este é um problema distinto de, por exemplo, uma seleção *kNN* incremental, visto que o interesse não está nos objetos que constituem o rol dos *kNN*.

Figura 4.1: *Wide-join* com predicados distintos: estrelas representam os elementos da primeira relação. Círculos são os elementos da segunda relação que satisfazem o predicado θ_s . Triângulos são os elementos da segunda relação que não atendem ao predicado. Losangos são os κ_f elementos selecionados pelo operador e pareados com as estrelas.

Nesse contexto, o controle da cardinalidade do resultado intrínseco ao operador *wide-join* permite empregar predicados por similaridade baseados em negação e retornar uma resposta cujo tamanho é determinado pelo usuário.

Seguindo as definições de predicados descritas na seção 4.3.1, a negação do caso onde $\theta_s = \text{TRUE}$ não é aplicável a qualquer operador, seja ele uma seleção/junção por similaridade ou um *wide-join*. Um predicado onde $\theta_s = \text{FALSE}$ sempre tem um conjunto vazio como resposta, e, nesse caso, não há elementos a serem retornados ou filtrados por alguma condição.

Para um θ_s composto pela negação do operador *similarity range*, ou seja, um *not-in-range*, o operador *wide-join* resultante seleciona os pares de elementos $\langle r, s \rangle$ onde $d(r.A_1, s.A_2) > \xi$. Para um operador de junção, esta condição apresenta a limitação de retornar um conjunto de altíssima cardinalidade. Uma forma de resolver este problema para o operador *range join* é aumentar o valor do limiar ξ para obter um conjunto reduzido de elementos. Contudo, conforme ξ cresce, os elementos recuperados são aqueles presentes nas regiões mais afastadas do espaço de busca, o que não condiz com os conceitos de similaridade e negação, visto que os elementos recuperados seriam dissimilares do centro de consulta. Assim, predicados baseados no operador *not-in-range* são processados através do operador *wide-join*, sendo que o limite superior de cardinalidade κ_f garante a recuperação de pares de elementos que estejam além do limiar ξ e ainda evita que o operador retorne todo o conjunto de elementos presentes na base de dados.

O pseudocódigo apresentado no algoritmo 4.1 permite computar o resultado do operador *not-in-range wide-join* através de uma estratégia de laço aninhado. O passo 4 realiza a comparação do tipo *not-in-range*. Enquanto o conjunto-resposta T contém menos que κ_f pares de elementos, as tuplas cujo valor de dissimilaridade excedem o limiar ξ são temporariamente incluídas no resultado (passos 5-6). Assim que T possui κ_f pares, os passos 9-11 substituem o par mais dissimilar do resultado pelo par mais similar, atualmente analisado. O condicional do passo 5 garante no máximo κ_f pares no conjunto-resposta, impedindo que o predicado *not-in-range* retorne todo o conjunto de dados. Uma vez que esta construção é dependente do parâmetro κ_f , esta solução é específica para o operador *wide-join*.

A figura 4.1(g) mostra o resultado de um *wide-join* empregando o operador de comparação *not-in-range*. Como ilustrado, os elementos de segunda relação dentro da região delimitada pelo raio ξ não fazem parte da resposta (triângulos). Entre os elementos que estão fora destas fronteiras delimitadas, ou seja, o restante do conjunto de dados, apenas aqueles que são os κ_f mais próximos dos elementos da primeira relação (estrelas) são combinados e retornados pelo operador, enquanto que os demais (círculos) são descartados.

De modo semelhante, o comparador *not-the-k-nearest* compondo o predicado θ_s resulta no operador *not-k-nearest neighbor wide-join*. Este operador recupera os pares de elementos mais similares que estão além dos k pares obtidos em um *kNN join*, mas restrito ao limite de κ_f pares. A implementação deste operador é semelhante à apresentada no algoritmo 4.1, variando apenas três pontos chaves: (i) a condição expressa no passo 4 é suprimida; (ii) a cardinalidade do passo 5 é verificada como sendo menor que $(\kappa_f + k)$; e (iii) para cada $r \in R$, T torna-se um conjunto-resposta temporário $T_{temp_i, r}$ e a função devolve um conjunto final $T = T_{temp_1} \cup T_{temp_2} \cup \dots \cup T_{temp_n}$.

A figura 4.1(j) ilustra o operador *not-k-nearest neighbor wide-join*. Para este operador, os pares de elementos que compõem a resposta de um *kNN join* não são considerados como possíveis candidatos a comporem o resultado final, mas apenas aqueles que estão além da vizinhança k . Assim, somente os κ_f pares mais semelhantes integram o conjunto-resposta.

Algoritmo 4.1: notInRangeWideJoin($R, S, \xi, \kappa_f, d()$)**Entrada:** R, S : os conjuntos de tuplas de entrada. ξ : limiar de similaridade. κ_f : parâmetro da consulta. $d()$: a função de distância.**Saída** : T : o conjunto-resposta.

```

1  $T \leftarrow \emptyset$ ;
2 para cada  $r \in R$  faça
3   para cada  $s \in S$  faça
4     se  $d(r, s) > \xi$  então
5       se  $|T| < \kappa_f$  então
6          $T \leftarrow T \cup \{ \langle r, s \rangle \}$ ;
7       senão se  $d(r, s) < \max_{\langle a, b \rangle \in T} d(a, b)$  então
8          $T \leftarrow T - \{ \argmax_{\langle a, b \rangle \in T} d(a, b) \}$ ;
9          $T \leftarrow T \cup \{ \langle r, s \rangle \}$ ;
10      fim
11    fim
12  fim
13 fim
14 retorna  $T$ ;

```

Ambas as negações dos operadores de comparação por similaridade podem ser combinadas com as suas versões “diretas” para comporem predicados compostos por mais de um termo, como discutido na próxima seção 4.3.3.

4.3.3 Predicados de termos múltiplos

Nas seções 4.3.1 e 4.3.2 foram apresentados quatro operadores de comparação baseados em similaridade. Entretanto, assume-se que um predicado por similaridade esta na forma $\theta_s = \varphi \tau_1 \varphi \tau_2 \varphi \dots \varphi \tau_n$, onde φ é um conectivo lógico de conjunção ($\wedge$), disjunção ($\vee$) ou negação ($\neg$), e τ é um termo que faz uso dos comparadores previamente apresentados.

Como pode ser notado, esta representação de predicados por similaridade faz com que a quantidade de instâncias distintas do operador *wide-join* seja não-enumerável. Além disso, devido à combinação de vários termos por similaridade, muitas dessas instâncias podem vir a ser equivalentes a outras. Contudo, como os múltiplos termos do predicado são relacionados de modo conjuntivo ou disjuntivo, cada termo τ pode ser processado separadamente de modo que cada resultado individual seja combinado aos demais através das operações de união e intersecção de conjuntos, no lugar dos operadores $\vee$ e $\wedge$, respectivamente. Nota-se, entretanto, que caso o operador unário de negação $\neg$ apareça na composição de um predicado, este tem prioridade de processamento sobre os conectivos $\vee$ e $\wedge$, enquanto que o operador de conjunção também tem prioridade sobre o operador de disjunção.

Como exemplo, as figuras 4.1(b), 4.1(e), 4.1(h) e 4.1(k) ilustram operadores *wide-join* compostos por predicados conjuntivos de dois termos, sendo um deles um *kNN* e outro um *range*, e seus comple-

Tabela 4.1: Configuração paramétrica dos experimentos: valores *default* parentizados em negrito

Parâmetro	Conjunto de dados	Valores
R	UniProt	(861) - proteínas presentes em humanos
	Sintético	(1000) ; 2500; 5000; 7500; 10000
S	UniProt	(12005) - proteínas não presentes em humanos
	Sintético	1000; 2500; 5000; 7500; 10000; (100000)
ξ	UniProt	(1) ; 2; 3; 4; 5
	Sintético	0,25; 0,50; 0,75; (1,00) ; 1,25
k	(Todos)	(5) ; 10; 15; 20; 25
κ_f	(Todos)	5; (10) ; 15; 20; 25
Dimensão	Sintético	2; (4) ; 8; 16; 32; 64
Distribuição	Sintético	normal; (uniforme)

mentos. Nota-se que o operador *k-and-range wide-join* (figura 4.1(b)) corresponde à generalização do operador *join around* (seção 2.4), para admitir valores de $k > 1$.

De modo análogo, as figuras 4.1(c), 4.1(f), 4.1(i) e 4.1(l) ilustram predicados binários com disjunção, isto é, compostos por dois termos relacionados pelo operador lógico OR. Nota-se que este tipo de conector possui menor seletividade que o conector conjuntivo, e por isso ele seleciona um maior número de candidatos (círculos).

4.4 Experimentos

Este capítulo descreve a extensão do operador de junção por similaridade em um conjunto mais amplo de operadores binários que empregam os conceitos de ordenação internamente, possibilitando ampliar a sua aplicação em consultas por similaridade. Para a avaliação do operador *wide-join* com seus distintos predicados, foram realizados testes de escalabilidade das instâncias obtidas, onde alterou-se as cardinalidades das relações, a distribuição dos dados no espaço, o número de dimensões dos conjuntos de dados e os parâmetros de consulta ξ , k e κ_f .

Os resultados descritos foram obtidos a partir de diversos conjuntos de dados sintéticos com diferentes valores de cardinalidades e dimensionalidades, além do conjunto de dados UniProt, já apresentado na seção 3.7.

Os conjuntos sintéticos variam de 1000 a 100000 pontos, em 2, 4, 8, 16, 32 e 64 dimensões, sendo que cada configuração foi gerada tanto para as distribuições Normal quanto Uniforme. Para o conjunto de proteínas, foram selecionadas 12866 cadeias cujo comprimento varia de 5 a 15 aminoácidos. Os pontos dos conjuntos sintéticos foram comparados através de métrica L_2 , enquanto que as proteínas foram comparadas pela distância de edição.

Os experimentos foram executados em um computador com processador Intel® Core™ i7-4770, 3.4 GHz, com 16 GB de RAM e sistema operacional GNU Linux, distribuição Ubuntu 14.04. Todos os algoritmos foram implementados em C++ usando o mesmo *framework*, para que pudessem ser comparados. As tuplas de todas as relações das junções permaneceram armazenados em disco,

de modo que as tuplas foram carregadas em memória primária apenas quando necessárias. Os resultados obtidos são mostrados na figura 4.2, enquanto que a tabela 4.1 mostra a variação dos parâmetros para cada conjunto de dados, tendo os valores padrão (*default*) de cada variável entre parênteses em negrito.

A figura 4.2(a) mostra o desempenho do operador *range wide-join* e de sua versão algébrica, respectivamente, os primeiro e segundo membros da equação 4.2, quando a cardinalidade do conjunto sintético aumenta. Como pode ser observado, para uma cardinalidade de 1000 elementos, o operador *range wide-join* (Rng) é, no mínimo, 6,62% mais rápido que a composição algébrica (AlgRng). Contudo, o ganho de desempenho aumenta conforme a cardinalidade do conjunto aumenta. De fato o melhor ganho encontrado corresponde à cardinalidade de 10000 elementos, onde o Rng foi 71,75% mais rápido que o AlgRng.

A figura 4.2(a) também compara o *range wide-join* com o operador *range join* (SimRng), presente na literatura. Como esperado, o SimRng é mais rápido que o Rng, visto que o segundo, além de processar internamente o primeiro, também realiza operações adicionais. No entanto, o SimRng foi, em média, apenas 6,87% mais rápido que o Rng.

Na figura 4.2(b) é apresentada uma análise similar à da figura 4.2(a), mas considerando o operador de comparação por similaridade *k-nearest neighbor*. O *k-nearest neighbor wide-join* (KNN) foi, em média, 76,78% mais rápido que sua versão algébrica (AlgKNN) com relação ao aumento do tamanho das relações de entrada. O *k-nearest neighbor wide-join* foi também comparado com o *k-nearest neighbor join* (SimKNN), sendo que este último obteve um tempo de processamento 14,25% menor que o primeiro, em média.

As figuras 4.2(a) e 4.2(b) mostram que, embora os operadores propostos não sejam mais rápidos que os seus operadores de junção por similaridade equivalentes, o processamento adicional introduzido para filtrar os pares mais similares não influencia na complexidade computacional.

A figura 4.2(c) ilustra o desempenho do operador *wide-join* empregando a negação dos predicados *range* e *kNN*. Os operadores resultantes, NotRng e NotKNN, executaram em tempo similar, visto que eles compartilham a mesma estrutura de implementação (algoritmo 4.1), sendo o NotRng 9,45% mais rápido que o NotKNN, em média. Ambos os operadores NotRng e NotKNN foram, respectivamente, 63,86% e 92,62% mais rápido que suas versões algébricas. Nota-se que não existe algoritmo especializado para executar nenhum deles na literatura.

Adicionalmente, verifica-se que a negação dos predicados por similaridade (figura 4.2(c)) segue o mesmo comportamento das versões tradicionais de junção, em que o operador de comparação *range* produz operadores mais rápidos que o *kNN*, como pode ser observado comparando as figuras 4.2(a) e 4.2(b). Observando os resultados obtidos nas figuras 4.2(a), 4.2(b) e 4.2(c), é possível notar que a complexidade computacional dos operadores é, como previsto na teoria, proporcional a $O(|R| \cdot |S|)$, assumindo o formato de parábola.

A figura 4.2(d) mostra o número de pares de elementos retornados pelas junções por similaridade *range* e *k-nearest neighbor*, internas ao operador *wide-join*, antes da realização das operações subsequentes de ordenação e filtragem. Este teste confirma que ambos os operadores de junção por similaridade recuperam mais pares do que o usuário está em geral interessado ou tem condições

Figura 4.2: Avaliação da escalabilidade do *wide-join*

de avaliar, os quais, na maioria dos casos, acabam sendo descartados em operadores subsequentes dos comandos de consulta, o que representa desperdício de processamento e recursos computacionais. De modo distinto a estes operadores, que trouxeram uma quantidade da ordem de dezenas de milhares de elementos, as instâncias do *wide-join* devolveram sempre apenas 10 pares em cada execução sobre o conjunto de dados sintéticos.

A figura 4.2(e) apresenta o efeito do aumento da dimensionalidade no desempenho dos operadores considerados. Como pode ser observado neste teste, a diferença entre os operadores é pequena. Entretanto, conforme o número de dimensões aumenta, a métrica torna-se computacionalmente mais custosa e o desempenho diminui.

Na figura 4.2(f) é mostrado o desempenho dos operadores Rng, KNN, NotRng e NotKNN sobre conjuntos sintéticos bidimensionais gerados a partir das distribuições Uniforme e Normal. O tempo de execução do operador *wide-join* para ambas as distribuições permaneceu idêntico, mostrando que *wide-joins* não são afetados pela distribuição dos dados. Uma vez que o algoritmo interno de junção é um laço aninhado, todos os elementos do espaço serão verificados, independentemente de sua localização, isto é, a distribuição do conjunto.

Na sequência, as figuras 4.2(g), 4.2(h) e 4.2(i) estudam os efeitos da variação dos parâmetros de consulta ξ , k e κ_f sobre o conjunto de dados UniProt. Nas figuras 4.2(g), 4.2(h), pode-se observar que conforme o valor do limiar de similaridade ξ e da constante k aumentam, mais elementos são combinados na fase de junção e incluídos no conjunto-resposta, o que faz com que o desempenho da fase de ordenação subsequente seja reduzido, visto que há mais elementos a serem processados.

Na figura 4.2(i), o aumento do parâmetro κ_f produz um resultado com uma quantidade maior de pares que são obtidos da fase de ordenação anterior. Contudo, filtrar mais ou menos pares conforme o valor de κ_f varia é realizado em tempo computacional linear, proporcional ao tamanho do conjunto. Assim, a variação de κ_f não afeta o desempenho geral do operador *wide-join*.

4.5 O operador *self wide-join*

O operador *wide-join* foi concebido para processar a junção entre duas relações distintas e devolver apenas as combinações dos objetos mais similares em geral. Contudo, como visto nos capítulos 1 e 2, diversas tarefas que empregam os operadores de junção por similaridade são realizadas sobre um repositório de dados que consiste em um único conjunto de dados. Dessa forma, estas operações envolvem combinar os elementos de um conjunto com eles próprios, o que é chamado de uma auto-junção (*self-join*).

Entretanto, a definição de *wide-join* apresentada na seção 4.2 não é adequada para o processamento de auto-junções. A razão é que, dada uma relação $R = \{r_1, \dots, r_n\}$, uma operação $R \bowtie R$ sobre qualquer predicado θ incluiria os κ_f pares de elementos cujas componentes possuem as menores distâncias entre si, o que, no caso, seriam os pares simétricos $\langle r_1, r_1 \rangle, \langle r_2, r_2 \rangle, \langle r_3, r_3 \rangle, \dots, \langle r_i, r_i \rangle$. Ou seja, o operador inclui na resposta a combinação de cada elemento com si próprio, de acordo com a propriedade da identidade dos indiscerníveis. Isso distorce a quantidade de pares obtidos na resposta em qualquer operador *wide-join* que tenha algum parâmetro baseado em contagem.

A fim de permitir a realização de “*wide-joins*” com o mesmo conjunto de dados, o operador *wide-join* foi estendido para uma nova variante, chamada de *self wide-join*. Essa variante executa: (i) a avaliação da operação por similaridade entre o *mesmo* conjunto ou relação e (ii) recupera os pares de elementos mais similares em geral. Formalmente, o operador *self wide-join* é apresentado pela definição 16.

Definição 16 (*Self wide-join*). *Seja $\mathcal{S}$ um universo de dados complexos, R uma relação, $A \subseteq R$ um conjunto de atributos complexos no domínio $\mathcal{S}$, $d : \mathcal{S} \times \mathcal{S} \mapsto \mathbb{R}_+$ uma função de distância sobre $\mathcal{S}$ e $\kappa_f \in \mathbb{N}^*$ uma constante não-nula. Um *self wide-join* $\bowtie_{(A, \theta_s, \kappa_f)} R$ é um operador de junção onde ambas as relações de entrada são a mesma relação R . O operador retorna os κ_f pares $\langle r_1, r_2 \rangle \mid r_1, r_2 \in R, r_1 \neq r_2$ mais similares entre si. O *self wide-join* é expresso em álgebra relacional de acordo com a equação 4.3.*

$$\bowtie_{(A, \theta_s, \kappa_f)} R \equiv \sigma_{(ord \leq \kappa_f)} \left(\Pi_{\{R, S, \mathcal{F}(d(A_1, A_2)) \rightarrow ord\}} \left(\rho_{(A/A_1)}(R) \stackrel{((A_1 \neq A_2) \wedge (A_1 \theta_s A_2))}{\bowtie} \rho_{(A/A_2)}(R/S) \right) \right) \quad (4.3)$$

4.5.1 Um algoritmo para *self range wide-joins*

De forma análoga ao *wide-join*, o operador *self wide-join* pode ser expresso usando os mesmos predicados apresentados na seção 4.3. Quando $\theta_s = (A_1 \text{ kNN}(d, k) A_2)$ tem-se um *self k-nearest neighbor wide-join*, e qualquer método apresentado no capítulo 3 pode ser empregado para computar a junção por similaridade interna. Entretanto, para um predicado $\theta_s = (A_1 \text{ Rng}(d, \xi) A_2)$, que produz um *self range wide-join*, um algoritmo baseado em pivôs foi especialmente projetado.

O algoritmo desenvolvido foi chamado de *Pivot-based Self Wide-Join* (PSWJ). Ele tem como base a propriedade da desigualdade triangular e a utilização de elementos de referência para realizar podas no espaço de busca e reduzir a quantidade de cálculos de distância, o que resulta em melhor desempenho.

Para um número m arbitrário de pivôs, $m \ll n, n = |R|$, escolhidos entre os elementos disponíveis no conjunto de dados, primeiramente, computa-se a distância entre cada elemento $r \in R$ para cada pivô $p_1, \dots, p_m$, sendo que a ideia é que cada elemento da base de dados seja filtrado de acordo com a sua distância ao conjunto de pivôs. Neste primeiro passo, $m \cdot n$ cálculos de distância precisam ser realizados.

O próximo passo corresponde à execução da junção por similaridade. Cada elemento r_i analisado no laço externo da junção é comparado com os elementos r_j do laço interno e, caso a distância $d(r_i, r_j) \leq \xi$, eles são combinados e adicionados ao conjunto-resposta. Entretanto, a fim de evitar comparações desnecessárias, primeiro verifica-se se cada r_j está na região coberta por r_i com relação aos pivôs, sendo que o cálculo de distância só é realizado se as condições expressas nas equações 4.4 e 4.5 forem simultaneamente satisfeitas para todos os pivôs p_k :

$$d(p_k, r_j) \geq d(p_k, r_i) - \xi \quad (4.4)$$

$$d(p_k, r_j) \leq d(p_k, r_i) + \xi \quad (4.5)$$

Usando como exemplo a figura 4.3, o elemento r_2 satisfaz as condições expressas nas equações 4.4 e 4.5 com relação ao pivô p_1 , i.e., r_2 está contido no hiper-anel delimitado pelo pivô p_1 . Entretanto, analisando em relação ao pivô p_2 , verifica-se que o elemento r_2 não satisfaz a condição expressa na equação 4.5. Portanto, uma vez que r_2 está fora da região de intersecção entre os dois hiper-ânéis projetados pelos pivôs p_1 e p_2 , é garantido que sua distância para o elemento r_i será maior que ξ , e ele pode seguramente ser descartado.

Ainda considerando a figura 4.3, é possível observar que o elemento r_1 satisfaz as equações 4.4 e 4.5 para ambos os pivôs, e, por isso ele deve ser comparado com r_i . Neste caso, embora r_1 esteja dentro da região do espaço que deve ser analisada, ele não está dentro do raio de abrangência de r_i , o que pode ser verificado com apenas um cálculo de distância.

Para os elementos dispostos dentro da região de análise, o número de comparações adicionais é baseado na distribuição dos dados e não pode ser estimado de antemão. Entretanto, o pior (e muito pouco provável) caso é quando todos os n elementos do conjunto de dados estiverem inseridos na região de análise. Neste caso, é necessário realizar, para cada elemento r_i , n cálculos de distância;

Figura 4.3: Estratégia baseada em pivôs para realizar podas no espaço de busca

totalizado $mn + n(n-1)/2$ comparações. Da mesma forma que em um algoritmo de laço aninhado, devido à propriedade da simetria, apenas metade das comparações necessitam de ser realizadas, pois trata-se de um *self join*.

Entretanto, é extremamente incomum e fácil de evitar que todo o conjunto de dados esteja disposto na região de análise. Na figura 4.3, fazendo $m = 3$ e adicionando o terceiro pivô próximo ao elemento r_2 reduziria consideravelmente a região de análise, tornando-a quase que restrita à cobertura de r_i .

A situação oposta acontece quando nenhum elemento se qualifica para ser analisado. Neste caso, não há cálculos de distância adicionais ao total de $m \cdot n$ previamente realizados.

No caso médio, a quantidade de comparações pode ser estimada como a média entre os melhor e pior casos, levando a um número de cálculos de distância de $\left(nm + \frac{n(n-1)}{4}\right)$, onde já estão inclusos os cálculos de distâncias para os pivôs.

Um *pivot-based self wide-join* pode ser implementado em memória secundária seguindo a abordagem de laço aninhado em blocos, e que é apresentada pelo algoritmo 4.2. No passo 1, o algoritmo utiliza uma fila de prioridade a fim de realizar a ordenação inerente ao operador *wide-join*. No passo 2, são selecionados aleatoriamente m elementos do conjunto de dados para serem utilizados como pivôs. Os passos 3-6 iteram sobre os blocos onde as tuplas estão armazenadas. O laço aninhado dos passos 8-13 itera sobre os elementos contidos nos blocos. Para evitar combinar um elemento consigo mesmo, a condição do passo 10 realiza o incremento da posição inicial do laço interno de junção, com base na propriedade da simetria.

Para cada pivô selecionado no passo 2, as equações 4.4 e 4.5 devem ser satisfeitas ao mesmo tempo, como especificado no passo 14. Observa-se que as distâncias entre os elementos e os pivôs podem ser pré-computadas e armazenadas durante a leitura dos elementos no laço dos passos 8-13. A condição do passo 14 verifica se o elemento r_j encontra-se disposto dentro da região de análise, criada pela intersecção dos hiper-anéis definidos pelos pivôs. A pertinência do elemento r_j à região de abrangência de r_i é verificada nos passos 15-16, em que um cálculo de distância adicional é realizado. Nos passos 15-18, o algoritmo verifica se já existem κ_f elementos no conjunto-resposta Q , e em caso negativo, os pares obtidos são adicionados ao resultado. Se Q já contiver κ_f pares,

Algoritmo 4.2: `pivotBasedSelfWideJoin(R,  $\xi$ , m,  $\kappa_f$ , d())`**Entrada:** R: os conjuntos de objetos de entrada. ξ : limiar de similaridade.

m: o número de pivôs.

 κ_f : parâmetro da consulta.

d(): a função de distância.

Saída : Q: o conjunto-resposta.

```

1   $Q \leftarrow \emptyset$ ;
2   $P \leftarrow (m \text{ elementos aleatórios de } R)$ ;
3  para  $x \leftarrow 1$  até número de blocos em R faça
4      para  $y \leftarrow x$  até número de blocos em R faça
5          carrega o bloco  $x$  na memória;
6          carrega o bloco  $y$  na memória;
7           $i \leftarrow 1$ ;
8          enquanto  $i < \text{número de elementos no bloco } x$  faça
9               $j \leftarrow i$ ;
10             se  $x = y$  então
11                  $j \leftarrow j + 1$ ;
12             fim
13             enquanto  $j < \text{número de elementos no bloco } y$  faça
14                 se (4.4) E (4.5) são simultaneamente verdadeiras  $\forall p_k \in P$  então
15                      $dist \leftarrow d(r_i.A_1, r_j.A_2)$ ;
16                     se  $dist \leq \xi$  então
17                         se  $|Q| \leq \kappa$  então
18                              $Q \leftarrow Q \cup \{\langle \langle r_i, r_j \rangle, dist \rangle\}$ ;
19                         senão
20                              $q \leftarrow \operatorname{argmax}_{\langle \langle r_i, r_j \rangle, dist \rangle \in Q} (dist)$ ;
21                             se  $dist < q.dist$  então
22                                  $Q \leftarrow Q - \{q\}$ ;
23                                  $Q \leftarrow Q \cup \{\langle \langle r_i, r_j \rangle, dist \rangle\}$ ;
24                             fim
25                         fim
26                     fim
27                 fim
28             fim
29         fim
30     fim
31 fim
32 retorna Q;

```

os passos 19-23 verificam se o par analisado é mais similar que o par menos similar presente no conjunto-resposta, e, em caso afirmativo, realiza a substituição.

4.5.2 Aplicação do *self wide-join* para detecção de quase-duplicatas

Os operadores de junção por similaridade podem ser aplicados em diversas tarefas, incluindo a detecção de imagens quase-duplicadas. Para este último propósito, estes operadores foram explo-

rados apenas sobre dados representados por cadeias de caracteres, usando funções de distância baseadas em distância de edição.

A detecção de quase-duplicatas em repositórios de imagens é uma tarefa importante que permite refinar o resultado de uma consulta, visto que apresentar imagens muito similares reduz o potencial exploratório das consultas. Para outras aplicações, o interesse pode ser justamente a recuperação de duplicatas, sendo necessário excluir do resultado os elementos que apresentam elevada dissimilaridade. Desta forma, esta seção apresenta um modelo de *software* para *near-duplicate detection* que faz uso do operador *self range wide-join*.

O modelo para detecção de quase duplicatas é composto por dois módulos. O primeiro é o módulo de extração de características, que processa imagens de forma idêntica a um sistema de recuperação por conteúdo. Este módulo faz uso dos algoritmos existentes na biblioteca Arboretum¹, desenvolvida pelo Grupo de Bases de Dados e Imagens da Universidade de São Paulo. Formalmente, este módulo recebe um repositório de imagens $C = \{img_1, \dots, img_m\}$, com m imagens e extrai as características visuais $v_i = \mathfrak{F}(img_i)$ de cada imagem $img_i \in C$. Cada v_i é um vetor de características $\langle f_1, \dots, f_n \rangle$, onde n é o número de características extraídas pelo método extrator $\mathfrak{F}$. As características dependem do tipo de aspecto visual considerado, como cor, forma ou textura, como discutido no capítulo 2. O resultado é um conjunto de m vetores de características que constituem o atributo A da relação R a ser submetida à junção, sendo que outras informações pertinentes às imagens podem ser armazenadas em outros atributos.

O segundo módulo, que realiza a detecção das duplicatas, é o módulo principal do sistema. Este módulo executa o operador *self range wide-join* sobre R . Os dois parâmetros do operador, ξ e κ_f , permitem ajustar a maneira como as comparações são realizadas, de modo a ir ao encontro da forma como a percepção do usuário considera imagens como sendo duplicatas. O *software* retorna os pares de imagens resultantes, permitindo que as duplicatas sejam mantidas para análise ou removidas, de acordo com o interesse do usuário. O modelo proposto foi avaliado usando um conjunto de imagens que ilustram situações de emergência, onde é imperativo que resultados de qualidade sejam rapidamente gerados e que duplicatas sejam removidas, a fim de fornecer uma visão global da ocorrência.

4.5.3 Avaliação do operador *self range wide-join*

Os experimentos realizados para avaliar o operador *self range wide-join* empregaram um conjunto de dados real (Fire) composto por 272 imagens de incidentes envolvendo incêndio. Essas imagens foram obtidas em uma simulação de situação de emergência realizada em um complexo industrial. O conjunto de dados foi previamente rotulado por especialistas no domínio, sendo que 25 cenas distintas foram reconhecidas. As imagens foram submetidas ao extrator *Color Layout* [51], que gerou vetores de características com 16 dimensões, comparados pela métrica L_2 .

Foram executados dois algoritmos para computar o *self range wide-join*: o algoritmo 4.2 (PSWJ) e o algoritmo tradicional de laço aninhado (NLWJ – *Nested Loop Wide-Join*). Ambos foram comparados com o método chamado de *Adaptive Custers with k-Means* (ACMe) [56], que realiza a detecção

¹Arboretum: <http://gbdi.icmc.usp.br/en/projects/> - Acesso: 5/Abril/2018

Figura 4.4: Avaliação de dados puramente métricos – conjunto Fire

de imagens quase duplicadas e que, segundo seus autores, é superior às técnicas presentes na literatura. Visto que o algoritmo *k-means* é sensível a *outliers* e que os centroides computados não correspondem a imagens reais, foi gerado um terceiro algoritmo, a partir do ACMe, que substitui o *k-means* pelo método dos *k-medoids*, denotado por ACMd. Com relação ao parâmetro de consulta ξ , este foi configurado para recuperar 1% da quantidade total de pares possíveis. Os experimentos avaliaram o desempenho das quatro técnicas e a qualidade das respostas retornadas, como mostrado na figura 4.4.

A figura 4.4(a) apresenta o tempo de execução total de cada método avaliado. As medidas de tempo mostradas correspondem apenas à execução do módulo de detecção de quase-duplicatas, visto que o processo de extração de características foi realizado apenas uma vez e o resultado obtido é compartilhado pelos quatro algoritmos. O PSWJ foi 44,45% mais rápido que o NLWJ. Além disso, ambas as técnicas baseadas no operador *self range wide-join* foram 2 ordens de magnitude mais rápidas que o ACMe e 3 ordens de magnitude mais rápidas que o ACMd.

Este ganho de desempenho é explicado devido às técnicas ACMe e ACMd aplicarem o algoritmo de agrupamento e, para cada grupo obtido, reapplicarem o método recursivamente, até que um valor limiar calculado em seus processamentos seja alcançado. De maneira distinta, os algoritmos PSWJ e NLWJ realizam o processamento sem a necessidade de divisões sucessivas dos conjuntos obtidos. Além disso, os métodos ACMe e ACMd impõe que seja indicado um valor k que corresponde à quantidade de *clusters* prevista, mas que raramente é conhecida a priori. Porém, o resultado retornado possui mais que k grupos, devido às divisões sucessivas até que o valor limiar seja obtido.

A fim de analisar a qualidade das respostas, foram empregadas curvas de *Precision × Recall* ($P \times R$). Curvas de $P \times R$ são comumente utilizadas como métrica na avaliação de algoritmos de recuperação de informação. A precisão (*precision*) é definida como a taxa de elementos relevantes recuperados pelo número de elementos recuperados. A revocação (*recall*) é definida como a taxa de elementos relevantes recuperados pelo número de elementos do conjunto de dados. Em gráficos do tipo $P \times R$, quanto mais próxima uma curva está do topo, melhor é a técnica correspondente.

Figura 4.5: Quase duplicatas obtidas pelos algoritmos de junção

A figura 4.4(b) mostra as curvas de $P \times R$ obtidas pelos três métodos. Neste experimento, o resultado do algoritmo NLWJ não é mostrado, visto que ele é idêntico ao do PSWJ e, portanto, as curvas de qualidade geradas são idênticas, variando apenas o tempo de processamento. Entre os métodos comparados, o PSWJ alcançou o melhor valor de precisão para todos os valores de revocação, sendo, em média, 35,14% mais preciso do que o ACMe e 36,78% mais preciso que o ACMd. Após recuperar todas as imagens relevantes da base de dados (100% de revocação), o PSWJ obteve uma precisão de 66.00%, enquanto que a precisão máxima dos concorrentes foi de 12.50%.

A figura 4.5 mostra algumas imagens consideradas como duplicatas recuperadas por cada método. Novamente, não se mostrou o NLWJ, já que ele obtém a mesma resposta do PSWJ. Para uma imagem selecionada aleatoriamente como centro de consulta (*query sample* – figura 4.5(a): a 10ª imagem com rótulo 13), a figura 4.5(b) mostra as duplicatas retornadas pelo PSWJ. Como pode ser notado, este algoritmo recuperou somente as imagens corretas (mesmo rótulo) e que de fato são relacionadas à consulta, mesmo na presença de variação de escala ou de rotações.

As figuras 4.5(c) e 4.5(d) mostram os agrupamentos obtidos pelos métodos ACMe e ACMd, respectivamente, em que a mesma imagem de consulta (figura 4.5(a)) foi indicada. Como pode ser observado, ambos os métodos trouxeram falsos positivos, sendo que a existência de objetos errôneos no resultado contribui para a queda da medida de precisão. Em teoria, embora o ACMe produza grupos de qualidade inferior aos do ACMd, visto que os centroides não são imagens reais, enquanto que os medoides o são, a diferença de precisão entre eles foi, em média, de apenas 1,63%.

A melhor qualidade do *self range wide-join* quando comparado com as técnicas de *clustering* é explicada pelo fato dos últimos métodos produzirem grupos de acordo com centroides e medoides (*seeds*). Os demais elementos são então alocados aos grupos de acordo com a proximidade às *seeds*, mas o relacionamento de um elemento com o outro não é considerado. Deste modo, imagens dissimilares entre si, porém similares a uma *seed* são dispostas no mesmo grupo. Já o operador *self range wide-join* estabelece uma relação elemento-a-elemento (par), evitando este problema e a queda de precisão do resultado.

Para avaliar a escalabilidade do operador *self range wide-join* e verificar que o algoritmo 4.2 obtém melhor desempenho por reduzir a quantidade de comparações realizadas, o operador foi testado com outro conjunto de dados, o conjunto Aloí apresentado na seção 3.7. Nesse experimento foram

Figura 4.6: Avaliação da escalabilidade do *self range wide-join* – conjunto Aloï

aferidos os valores de tempo de execução e quantidade de cálculos de distância. Os resultados obtidos são apresentados na figura 4.6.

Na análise de escalabilidade mostrada na figura 4.6, as imagens do conjunto de dados foram embaralhadas e a cardinalidade da entrada submetida aos operadores foi gradativamente aumentada. A figura 4.6(a) mostra o tempo de execução demandado por cada método. O algoritmo PSWJ foi, em média, 49,59% mais rápido que o NLWJ. Além disso, o PSWJ foi, respectivamente, 2 e 3 ordens de magnitude mais rápido do que as técnicas ACMe e ACMd. Como exemplo, para o valor de cardinalidade de 10000, o PSWJ executou em 1,16 minutos, enquanto que o NLWJ consumiu 2,20 minutos, o ACMe levou 1,18 horas e o ACMd precisou de 6,61 horas.

A figura 4.6(b) compara os métodos PSWJ e NLWJ com relação à quantidade de cálculos de distância realizados para processar a operação de junção. O PSWJ executou pelo menos 44.69% menos comparações que o NLWJ para o valor de cardinalidade de 40K; mas a maior redução foi observada para a cardinalidade de 1K, onde o PSWJ realizou 68,75% menos cálculos. Contudo, é importante destacar que ambos os algoritmos obtiveram o mesmo resultado final, com a diferença de que o PSWJ foi capaz de poupar recursos computacionais em relação à versão tradicional de laço aninhado.

Otimizações semelhantes às realizadas no algoritmo 4.2 podem ser obtidas pelo estudo das propriedades algébricas entre os operadores da álgebra relacional e o *wide-join*. A seção 4.6 a seguir apresenta este estudo teórico.

4.6 Propriedades algébricas envolvendo *wide-joins*

Esta seção apresenta algumas propriedades algébricas do operador *wide-join*. Estas propriedades definem como o operador pode se relacionar com os demais operadores da álgebra relacional. Além disso, elas permitem realizar otimizações no plano de consulta e acelerar a geração de resultados. A seguir, são apresentadas as propriedades identificadas, enquanto que suas demonstrações estão no apêndice B. Para isso, são consideradas as relações R , S e T contendo, respectivamente, atributos

complexos $A_1 \subseteq R$, $A_2 \subseteq S$ e $A_3 \subseteq T$. Adicionalmente, para a demonstração da propriedade 3, são considerados a relação auxiliar U e o atributo complexo $A_4 \subseteq U$.

Para o desenvolvimento das demonstrações das propriedades identificadas, cada operação que compõe o operador *wide-join* foi escrita usando a notação equivalente da Teoria de Conjuntos. Dessa forma, o *wide-join* pode ser definido como segue:

Definição 17. *Wide-join: definição em Teoria de Conjuntos.*

$$R \stackrel{(A_1 \theta_s A_2), \kappa_f}{\bowtie} S = \underbrace{\sigma_{(ord \leq \kappa_f)}}_D \left(\underbrace{\Pi_{\{R, S, \mathcal{F}(dist) \rightarrow ord\}}}_C \left(\underbrace{\Pi_{\{R, S, d(A_1, A_2) \rightarrow dist\}}}_B \left(\underbrace{R \stackrel{(A_1 \theta_s A_2)}{\bowtie} S}_A \right) \right) \right)$$

Simplificando:

$$A = \{\langle r, s \rangle \in (R \times S) \mid r.A_1 \theta_s s.A_2\}$$

$$B = \Pi_{\{a_1, \dots, a_n, d(a.A_1, a.A_2) \rightarrow dist\}}(A) = \{t[a_1, \dots, a_n, dist] : t \in A\}$$

$$C = \Pi_{\{b_1, \dots, b_n, \mathcal{F}(b.dist) \rightarrow ord\}}(B) = \{t[b_1, \dots, b_n, ord] : t \in B\}$$

$$D = \{t \in C \mid t.ord \leq \kappa_f\}$$

Ou seja,

$$R \stackrel{(A_1 \theta_s A_2), \kappa_f}{\bowtie} S = \{t' \in \{t[r_1, \dots, r_n, s_1, \dots, s_n, ord] : t \in (R \times S) \mid t.A_1 \theta_s t.A_2\} \mid t'.ord \leq \kappa_f\} \quad (4.6)$$

□

A equação 4.6 fornece a base conceitual para o desenvolvimento das demonstrações das propriedades descritas a seguir.

A primeira propriedade identificada explora a comutatividade entre as relações do operador *range wide-join*, para qualquer limiar de similaridade ξ e qualquer quantidade κ_f dada. Esta propriedade é descrita como:

Propriedade 1 (Comutatividade de *range wide-join*). *Seja $\mathbb{S}$ um domínio de dados complexos, R e S duas relações, $A_1 \subseteq R$ e $A_2 \subseteq S$ dois atributos complexos amostrados em $\mathbb{S}$, $d : \mathbb{S} \times \mathbb{S} \mapsto \mathbb{R}_+$ uma função de distância, ξ um limiar de similaridade, e $\kappa_f \in \mathbb{N}^*$ uma constante não-nula. Então,*

$$R \stackrel{(A_1 \text{ Rng}(d, \xi) A_2), \kappa_f}{\bowtie} S = S \stackrel{(A_2 \text{ Rng}(d, \xi) A_1), \kappa_f}{\bowtie} R$$

É interessante observar que, embora válida, a propriedade 1 não segue a formalização estrita de relações definidas sobre o Modelo Relacional. Por exemplo, considerando a seguinte consulta

C1: recuperar os 10 pares de hotéis e pontos turísticos mais próximos entre si, a até 50 metros um do outro, onde o esquema da relação que armazena os hotéis é dado por $\text{Hotel} = \{\text{idH}, \text{nomeH}, \text{coordH}\}$ e o esquema da relação que guarda os pontos turísticos é dado por $\text{PTurist} = \{\text{idP}, \text{nomeP}, \text{coordP}\}$, em que ambos os atributos complexos coordH e coordP contêm a informação da coordenada geográfica (latitude e longitude) associada ao dado, têm-se que a expressão

$$\text{Hotel} \begin{matrix} (\text{coordH} \text{ Rng}(L_1, 50) \text{ coordP}), 10 \\ \bowtie \end{matrix} \text{PTurist}$$

devolve um conjunto de pares $\langle h, pt \rangle$ em que o primeiro componente corresponde a um hotel e o segundo componente é um ponto turístico. Por sua vez, a expressão

$$\text{PTurist} \begin{matrix} (\text{coordP} \text{ Rng}(L_1, 50) \text{ coordH}), 10 \\ \bowtie \end{matrix} \text{Hotel}$$

retorna pares de tuplas $\langle pt, h \rangle$ onde os dois componentes de cada par encontram-se invertidos. Embora esta inversão não obedeça à Teoria Relacional, onde a *ordem* dos atributos na tupla é importante e deve ser mantida, o *significado* associado à resposta da consulta (pares de hotéis e pontos turísticos) continua o mesmo.

A segunda propriedade identificada mostra que realizar a projeção dos atributos da relação-resposta de um *wide-join* é equivalente a projetar os atributos de cada relação original e realizar a operação de *wide-join* sobre essas projeções individuais. Esta propriedade é enunciada como:

Propriedade 2 (Distributividade de projeção sobre *wide-join*). *Seja $\mathbb{S}$ um domínio de dados complexos, R e S duas relações, $A_1 \subseteq R$ e $A_2 \subseteq S$ dois atributos complexos amostrados em $\mathbb{S}$, $d : \mathbb{S} \times \mathbb{S} \mapsto \mathbb{R}_+$ uma função de distância, θ_s um comparador por similaridade, e $\kappa_f \in \mathbb{N}^*$ uma constante não-nula. Então,*

$$\pi_{\{r_i, \dots, A_1, s_j, \dots, A_2\}} \left(R \begin{matrix} (A_1 \theta_s A_2), \kappa_f \\ \bowtie \end{matrix} S \right) = \left(\pi_{\{r_i, A_1\}}(R) \right) \begin{matrix} (A_1 \theta_s A_2), \kappa_f \\ \bowtie \end{matrix} \left(\pi_{\{s_j, A_2\}}(S) \right)$$

Como exemplo da propriedade 2 e considerando as relações de hotéis e pontos turísticos definidas, seja a consulta **C2:** recuperar o nome e a coordenada geográfica dos 10 pares de hotéis e pontos turísticos mais próximos entre si, a até 50 metros um do outro, que pode ser descrita em álgebra relacional como:

$$\pi_{\{\text{nomeH}, \text{coordH}, \text{nomeP}, \text{coordP}\}} \left(\text{Hotel} \begin{matrix} (\text{coordH} \text{ Rng}(L_1, 50) \text{ coordP}), 10 \\ \bowtie \end{matrix} \text{PTurist} \right)$$

Nota-se que em sua expressão equivalente, ou seja,

$$\left(\pi_{\{\text{nomeH}, \text{coordH}\}}(\text{Hotel}) \right) \begin{matrix} (\text{coordH} \text{ Rng}(L_1, 50) \text{ coordP}), 10 \\ \bowtie \end{matrix} \left(\pi_{\{\text{nomeP}, \text{coordP}\}}(\text{PTurist}) \right)$$

é necessário que os atributos complexos sobre os quais o operador *wide-join* realiza a comparação por similaridade (no exemplo, coordH e coordP) sejam projetados para ambas as relações de junção.

A terceira propriedade identificada discursa sobre a distributividade do operador *wide-join* sobre outro operador binário: a união de conjuntos. Esta propriedade é descrita como segue:

Propriedade 3 (Distributividade de *wide-join* sobre união). *Seja $\mathbb{S}$ um domínio de dados complexos, R , S e T três relações, $A_1 \subseteq R$, $A_2 \subseteq S$ e $A_3 \subseteq T$ três atributos complexos amostrados em $\mathbb{S}$, $A_4 \subseteq A_1 \cup A_2$ um atributo complexos, $d : \mathbb{S} \times \mathbb{S} \mapsto \mathbb{R}_+$ uma função de distância, θ_s um comparador por similaridade, e $\kappa_f \in \mathbb{N}^*$ uma constante não-nula. Então,*

$$T \overset{(A_3 \theta_s A_4), \kappa_f}{\bowtie} (R \cup S) \subseteq \left(T \overset{(A_3 \theta_s A_1), \kappa_f}{\bowtie} R \right) \cup \left(T \overset{(A_3 \theta_s A_2), \kappa_f}{\bowtie} S \right)$$

Ao contrário das propriedades 1 e 2, a propriedade 3 não apresenta uma relação de equivalência, e sim uma *relação de inclusão*, onde o resultado de um *wide-join* está contido no conjunto-resposta computado pela operação de união de outros dois *wide-joins*.

Como exemplo, além das relações de hotéis e pontos turísticos anteriormente definidas, seja uma relação que armazena as informações sobre restaurantes cujo esquema é dado por $\text{Rest} = \{\text{idR}, \text{nomeR}, \text{coordR}\}$, possuindo, também, um atributo complexo que armazena sua coordenada geográfica (coordR). Dessa forma, a expressão algébrica

$$\text{Hotel} \overset{(\text{coordH} \text{ Rng}(L_1, 50) \text{ coord}), 10}{\bowtie} (\text{PTurist} \cup \text{Rest})$$

que se traduz na consulta **C3**: *recuperar as 10 combinações de hotéis com pontos turísticos ou restaurantes mais próximos entre si, a até 50 metros um do outro*, está contida no resultado da seguinte consulta **C4**: *recuperar as 10 combinações de hotéis e pontos turísticos e as 10 combinações de hotéis e restaurantes mais próximos entre si, a até 50 metros uns dos outros*, algebricamente representada como:

$$\left(\text{Hotel} \overset{(\text{coordH} \text{ Rng}(L_1, 50) \text{ coordP}), 10}{\bowtie} \text{PTurist} \right) \cup \left(\text{Hotel} \overset{(\text{coordH} \text{ Rng}(L_1, 50) \text{ coordR}), 10}{\bowtie} \text{Rest} \right)$$

Nota-se que a expressão algébrica do primeiro membro da propriedade 3 considera que cada hotel deve ser pareado ou com um ponto turístico ou com um restaurante, enquanto a expressão algébrica do segundo membro realiza o pareamento dos hotéis com os pontos turísticos e também com os restaurantes, sem exclusão mútua entre essas localidades.

4.7 Considerações finais

Os operadores de junção por similaridade apresentam três limitações principais quando aplicados em ambientes relacionais: o volume de dados recuperado é normalmente alto, necessitando de etapas de filtragens (pós processamento) subsequentes; os elementos mais similares (objetivo de uma consulta por similaridade) encontram-se imersos no grande volume de objetos retornado; e eles demandam por operações de ordenação, um conceito não existente na teoria relacional.

Objetivando encontrar uma solução para essas limitações, este capítulo apresentou um conjunto de operadores binários por similaridade, chamados de *wide-joins*, que engloba o conceito de junção e produz resultados mais significativos que os operadores de junção por similaridade convencionais.

Os operadores *wide-joins* recuperam os pares de elementos mais similares num contexto geral, representando a ordem entre os elementos internamente, sem a necessidade de ordenar os dados de entrada ou saída, ou seja, de forma compatível com a teoria relacional. Foram apresentadas a definição algébrica e a especificação dos diferentes tipos de predicados suportados pelo *wide-join*, bem como a negação de comparadores por similaridade, que são exclusivos do operador proposto.

A avaliação experimental dos diversos operadores *wide-join* envolveu conjuntos de dados sintéticos distintos e um conjunto real, onde aspectos como a dimensionalidade, distribuição dos dados e escalabilidade foram verificados. Os experimentos mostraram que as instâncias de *wide-join* possuem desempenho equivalente ao dos operadores de junção por similaridade, retornando um resultado menor e de maior aplicabilidade e qualidade.

Finalmente, este capítulo apresentou uma versão do *wide-join* especialmente adaptada para tratar auto-junções, isto é, a combinação de um conjunto com ele próprio. Esta versão, o *self wide-join* foi aplicado na modelagem de um sistema para detecção de imagens quase-duplicadas. Por fim, foi identificado um conjunto de propriedades algébricas que facilitam a inter-operação do *wide-join* com os demais operadores da álgebra.

Conclusão

Os operadores de junção por similaridade possuem grande importância e aplicabilidade em um amplo domínio de tarefas. Entre os tipos de junção por similaridade existentes, o operador *range join* é o mais explorado pela literatura. Apesar da execução do operador *range join* ser a mais rápida dentre os operadores de junção, e para o qual foram propostas diversas otimizações, os operadores *k-nearest neighbor join* e *k-closest neighbor join* também possuem apelo prático, além de permitirem expressar consultas de maneira mais intuitiva que o *range join*. Contudo, apesar da versatilidade que possuem, os operadores de junção por similaridade apresentam diversas limitações. Estas limitações constituíram a principal fonte de motivação desta tese.

As oportunidades de exploração das soluções para os problemas relacionados aos operadores de junção por similaridade podem ser classificadas, basicamente, em duas linhas:

1. Desempenho do operador,
2. Qualidade do conjunto-resposta.

Com relação à primeira linha, os operadores de junção *kNN* e *kCN* são processados exclusivamente por técnicas computacionais de complexidade quadrática, sendo que as poucas oportunidades de otimização exploradas na literatura não são generalizáveis para todos os domínio de dados. Por causa dessas especificidades, essas técnicas são pouco adequadas para serem consideradas em um Sistema de Gerenciamento de Bases de Dados (SGBD), visto que um SGBD é essencialmente um *software* de uso geral. O estudo deste primeiro nicho foi o ponto de partida para o desenvolvimento da primeira contribuição desta tese: a proposta de um algoritmo eficiente para processamento de *kNN joins*.

Visando atacar também a segunda linha, uma segunda contribuição foi obtida a partir de um estudo algébrico preliminar sobre o operador *kCN join*. Esse estudo identificou que o operador binário *kCN* realiza a combinação de elementos através de critérios de similaridade, mas que vai além de ser apenas uma junção e envolve também uma ordenação e critérios adicionais de

filtragem. Dessa forma, a partir de sua definição inicial, o operador *kCN join* foi tomado como base para criar um novo conjunto de operadores binários que ampliam o conceito de junção por similaridade, que são capazes de avaliar predicados por similaridade e englobar o conceito de ordem internamente ao processamento. Este conjunto de operadores, que foram chamados de *wide-joins*, realizam a otimização do resultado inicial computado por uma junção interna, de modo a obter uma resposta mais precisa, de maior aplicabilidade e menor cardinalidade. A seguir, na seção 5.1, são detalhadas as contribuições desta tese.

5.1 Contribuições

Esta tese contribuiu para a otimização dos operadores de junção por similaridade explorando as necessidades de recuperação de dados em aplicações de análise de dados complexos comparados por similaridade.

Os objetivos inicialmente delineados foram consolidados na proposta de duas técnicas principais:

1. Um algoritmo para processamento de junções por similaridade empregando predicados do tipo *kNN*, e
2. Um conjunto de operadores binários que combina junção e ordenação para recuperação dos elementos mais similares num contexto global.

De modo geral, as contribuições alcançadas e seus desmembramentos podem ser sumarizados como:

1. Desenvolvimento do algoritmo *QuickNearest* para processar *k-nearest neighbor joins*: trata-se de uma abordagem de divisão e conquista extremamente eficiente, que chegou a ter desempenho 10 mil vezes (4 ordens de magnitude) melhor que os métodos que constituem o estado da arte, principalmente para dados de baixa dimensionalidade mas alta cardinalidade, e é mais rápida que os demais métodos em todas as situações.
 - Ao contrário das técnicas existentes na literatura, o *QuickNearest* é um algoritmo independente do domínio dos dados, ou seja, ele é capaz de processar qualquer tipo de dado (multidimensional ou adimensional), representados em espaços de buscas métricos ou não-métricos.
 - Foram descritas duas estratégias de particionamento do espaço de dados que podem ser combinadas com o algoritmo *QuickNearest*, o particionamento através de hiperplano generalizado e o particionamento por bolas.
 - Foi apresentada a análise de custo do algoritmo *QuickNearest* e das suas principais funções.
 - Foi realizada uma avaliação experimental extensiva do método proposto sob diversos aspectos, tais como a dimensionalidade dos dados, a distribuição dos dados no espaço, a variação paramétrica do algoritmo e o processamento de dados não-dimensionais.

2. Desenvolvimento do conjunto de operadores binários que combinam conceitos de junção por similaridade e ordenação de tuplas, a fim de recuperar um conjunto-resposta com melhor qualidade e de tamanho reduzido, os *wide-joins*.
 - O operador *k-closest neighbor join* foi apresentado como um operador binário estendido, que não segue a formalização do conceito de junção.
 - Os operadores de junção foram definidos sobre relações, tuplas e atributos, seguindo uma abordagem relacional e que pode ser acoplada a um SGBD. As definições presentes na literatura definem operadores de consulta por similaridade sobre conjuntos, onde não existe o conceito de separação de atributos.
 - O conceito de ordenação foi incluído em operadores de consulta por similaridade de modo compatível com a teoria relacional.
 - As definições algébricas do conjunto de operadores denotados por *wide-joins*, bem como propriedades algébricas relacionando-os com os demais operadores da álgebra relacional, foram desenvolvidas.
 - Foram analisados predicados específicos para o operador *wide-join*, incluindo aqueles que envolvem a negação de comparadores por similaridade.
 - Foi proposta uma variante do operador *wide-join* para a junção de uma relação com ela mesma, o operador unário *self wide-join*, bem como um algoritmo eficiente, baseado em particionamento do espaço por pivôs, para seu processamento.
 - Foi realizada a avaliação do *wide-join* e do *self wide-join*, mostrando que os operadores são escaláveis, computam um conjunto-resposta com precisão em geral 50% maior que os métodos concorrentes, e mostrou-se sua aplicação em situações reais, como, por exemplo, na detecção de quase-duplicatas.

Como trabalhos subsequentes às contribuições obtidas, destacam-se:

- A fusão do algoritmo *QuickNearest* com estruturas métricas de indexação.
- A criação de um algoritmo baseado em divisão e conquista, tal como o *QuickNearest*, para processar os operadores *k-closest neighbor join* e *wide-join*.
- A paralelização do algoritmo *QuickNearest*. Inicialmente, almeja-se implementar a divisão das funções do operador para mais de um *core* e, posteriormente, em rede, usando mais de um *host*.
- O desenvolvimento de modelos matemáticos para o *QuickNearest*, relacionando os valores de k e $|R|$ para estimar o limiar de sublimação H .
- A investigação de estratégias adaptativas para descoberta do intervalo ótimo para o limiar de sublimação H .
- O estudo de estatísticas e estimativas de seletividade de dados para permitir o ajuste dos parâmetros empregados no *wide-join*, tais como ξ , k e κ_f .

- A proposição e avaliação da representação dos operadores *wide-join* em SQL, considerando principalmente como associar a representação das condições e parâmetros do operador com as demais condições e argumentos usados para os demais operadores de um comando de consulta.
- A identificação de propriedades algébricas adicionais para o operador *wide-join*. O desenvolvimento das propriedades descritas no capítulo 4 constituíram a maior fonte de dificuldade desta tese, como relatado na próxima seção 5.2.

5.2 Dificuldades encontradas

Em geral, durante a formação discente e também docente na área de computação, a matemática é fortemente apresentada como uma disciplina essencialmente computacional. Aprende-se a *resolver* sistemas, *calcular* limites, derivadas, realizar substituições para *solucionar* integrais, *multiplicar* vetores, matrizes e *calcular* determinantes, gradientes etc; e, além disso, leva-se os aprendizes a entender como esses tópicos podem ser *aplicados* para responder questões práticas do mundo real.

Entretanto, há um outro lado da matemática que é mais teórico do que prático, cujo principal objetivo é *entender* as estruturas matemáticas, *provar* formulações e *descobrir* e *desenvolver* novas teorias ou teoremas matemáticos. Embora os procedimentos e técnicas matemáticas que aprendidos em cursos de graduação sejam fundamentados nesse lado mais teórico, a formação em computação não costuma explorá-los em profundidade. Por exemplo, para computar a área sob uma curva, recaímos no Teorema Fundamental do Cálculo porque este teorema é verdadeiro e sua resposta é correta. Entretanto, ainda na graduação, quando o cálculo é apresentado, foca-se mais em como o teorema é aplicado do que em entender de onde surgiu o teorema e o porquê ele é verdadeiro, o que, futuramente, leva a questões do tipo “*Como sabemos (temos convicção) que o teorema é verdadeiro?*” ou “*Como podemos convencer a nós mesmos e aos outros da validade do teorema?*”. Questões como estas são respondidas pelo lado de caráter teórico da matemática e envolveram a maior dificuldade desta tese.

A seção 4.6, apresentou o desenvolvimento de relações de equivalência e propriedades sobre como o operador *wide-join* se relaciona com os demais operadores da álgebra relacional. Entretanto, uma vez que a álgebra relacional e os operadores que sobre ela atuam são inteiramente fundamentados na Teoria de Conjuntos, o levantamento das propriedades já identificadas demandam por maior rigor matemático a fim de constatar sua validade. Assim, para provar a corretude de cada propriedade identificada, foi necessário recorrer a técnicas de *demonstração matemática*.

O primeiro trabalho que também fez uso do mesmo artifício é descrito em [34], porém, este trabalho introduz propriedades para operadores unários, enquanto que a presente tese envolveu operadores binários. Além disso, o estudo descrito em [34] é superficial em fornecer fundamentos matemáticos necessários e suficientes para a demonstrações mais gerais. Desse modo, foi necessário o estudo desse lado mais teórico da matemática através das seguintes referências:

- HALMOS, P. R. *Naive Set Theory*, Springer, 1974.

- ROSEN, K. H. *Discrete Mathematics and its Applications*, 7 ed. McGraw-Hill, 2011.

Embora as referências estudadas permitam resgatar conceitos de matemática discreta e da teoria de conjuntos, elas abordam a questão de demonstrações de modo mais direcionado ao cálculo algébrico, raramente enfatizando aspectos da teoria de conjuntos, e quando o fazendo, restringindo-se a propriedades elementares. Desse modo, foi necessário investigar esse aspecto específico de demonstrações matemáticas, com ênfase na teoria e álgebra de conjuntos. Para isso, foram estudadas partes das seguintes obras:

- FIELDS, J. *A Gentle Introduction to the Art of Mathematics*, 3 ed. 2015.
- HAMMACK, R. *Book of Proof*, 2 ed. 2013.
- SUNDSTROM, T. *Mathematical Reasoning: Writing and Proof*, 2 ed. 2015.

Entretanto, essas referências abordam a formalização matemática apenas para teoria de conjuntos, raramente tratando de relações seguindo as definições embasadas no Modelo Relacional. Além disso, a álgebra relacional é composta de seu próprio conjunto de operadores, sendo que cada um deles traduz uma representação específica da teoria de conjuntos. Assim, de modo a combinar o rigor matemático adquirido pelo estudo das obras acima citadas com aspectos da teoria relacional, a seguinte referência foi uma das mais valiosas:

- MAIER, D. *The Theory of Relational Databases*, Computer Science Press, 1983.

Outra dificuldade encontrada foi com relação à codificação do algoritmo *QuickNearest* (seção 3.3). Por se tratar de um método que explora a recursividade, seu processo de *debug* foi tortuoso e tartáreo. Conforme a cardinalidade do conjunto submetido ao método aumentava, o código passava a apresentar algumas instabilidades, como erros de violação e perda de referência de memória (*segmentation fault* e *memory leak*). Entretanto, dependendo do tipo de particionamento de espaço selecionado e da ordem de entrada dos dados no momento de execução do algoritmo, os erros relatados não foram observados, o que dificultou ainda mais sua descoberta.

Além disso, a variação do algoritmo rotulada por QNB apresentou, em alguns casos, resultados não coincidentes com os resultados do método NLJ. Por exemplo, na execução de um *kNN join* com cardinalidade de cinquenta mil elementos, sendo $k = 10$ (o que leva a um conjunto resposta com meio milhão de pares), o algoritmo QNB perdia 6 pares, isto é, o algoritmo substituiu 6 dos pares que estariam entre os 10 vizinhos mais próximos por pares de maior dissimilaridade. Em outras palavras, recuperou o 11-NN ou 12-NN ao invés do 10-NN em 6 pares. Descobrir onde estão e o porquê 6 pares no meio de meio milhão foram substituídos demandou enorme esforço de *debug*. O problema estava no método `partition()` (em ambos os algoritmos 3.5 e 3.8), no momento da verificação e alocação do último elemento testado em uma das duas partições geradas. A versão inicialmente desenvolvida desses métodos não continha a verificação condicional encontrada nos passos 24-28, fazendo com que alguns elementos não fossem corretamente alocados nas partições e ocasionando perdas.

5.3 Publicações

Durante o período de doutorado, foram obtidas 12 publicações em eventos nacionais e internacionais, sendo que 7 foram escritas pelo autor, onde ele aparece como primeiro autor, e as outras 5 foram desenvolvidas em parceria/colaboração com outros integrantes do mesmo grupo de pesquisa. Um outro artigo ainda está em fase final de redação para submissão. Em ordem cronológica, são elas:

- **CARVALHO, L. O.**, OLIVEIRA, W. D., POLA, I. R. V., TRAINA, A. J. M., AND TRAINA JR., C. A 'Wider' Concept for Similarity Joins. *Journal of Information and Data Management*, n. 5, v. 3 (Oct. 2014), 210–223.
- BEDO, M. V. N., **CARVALHO, L. O.**, PIERRO, G. V., AND TRAINA JR., C. SHRuB: Searching through Heuristics for the Better Query Execution Plan. In *Proceedings of the 29th Brazilian Symposium on Databases* (Curitiba, PR, Brasil, 2014), Brazilian Computer Society, pp. 245–250.
- **CARVALHO, L. O.**, SERAPHIM, E., SERAPHIM, T. F. P., TRAINA, A. J. M., AND TRAINA JR., C. Me-dInject: a General Purpose Information Retrieval Framework Applied in a Medical Context. In *Proceedings of the 27th IEEE International Symposium on Computer-Based Medical Systems* (New York, NY, USA, 2014), IEEE Computer Society, pp. 308–313.
- **CARVALHO, L. O.**, SANTOS, L. F. D., OLIVEIRA, W. D., TRAINA, A. J. M., AND TRAINA JR., C. Similarity Joins and Beyond: An Extended Set of Binary Operators with Order. In *8th International Conference on Similarity Search and Applications* (Glasgow, Scotland, 2015), vol. 9371 of *Lecture Notes in Computer Science*, Springer, pp. 29–41.
- **CARVALHO, L. O.**, SANTOS, L. F. D., OLIVEIRA, W. D., TRAINA, A. J. M., AND TRAINA JR., C. Self Similarity Wide-Joins for Near-Duplicate Image Detection. In *IEEE International Symposium on Multimedia* (Miami, FL, USA, 2015), pp. 237–240.
- SANTOS, L. F. D., **CARVALHO, L. O.**, OLIVEIRA, W. D., TRAINA, A. J. M., AND TRAINA JR., C. Diversity in Similarity Joins. In *8th International Conference on Similarity Search and Applications* (Glasgow, Scotland, 2015), vol. 9371 of *Lecture Notes in Computer Science*, Springer, pp. 42–53.
- SANTOS, L. F. D., OLIVEIRA, W. D., **CARVALHO, L. O.**, FERREIRA, M. R. P., TRAINA, A. J. M., AND TRAINA JR., C. Combine-and-Conquer: Improving the Diversity in Similarity Search Through Influence Sampling. In *30th ACM Symposium on Applied Computing* (Salamanca, Spain, 2015), pp. 994–999.
- **CARVALHO, L. O.**, SANTOS, L. F. D., OLIVEIRA, W. D., TRAINA, A. J. M., AND TRAINA JR., C. Efficient Self-Similarity Range Wide-Joins Fostering Near-Duplicate Image Detection in Emergency Scenarios. In *18th International Conference on Enterprise Information Systems* (Rome, Italy, 2016), vol. 1, Scitepress, pp. 81–91. **BEST STUDENT PAPER AWARD**
- SANTOS, L. F. D., **CARVALHO, L. O.**, BEDO, M. V. N., TRAINA, A. J. M., AND TRAINA JR., C. When Similarity is Not Enough, Ask for Diversity: Grouping Elements Based on Influence. In *IEEE International Symposium on Multimedia* (San Jose, CA, USA, 2016), pp. 26–29.

- TRAINA, A. J. M., BEDO, M. V. N., SANTOS, L. F. D., **CARVALHO, L. O.**, PEDROSA, G. V., COSTA, A. F., AND TRAINA JR., C. Making Content-Based Medical Image Retrieval Systems worth for Computer-Aided Diagnosis: From Theory to Application. In *Medical Image Analysis and Informatics: Computer-Aided Diagnosis and Therapy*, P. M. A. Marques, A. Mencattini, M. Salmeri, and R. M. Rangayyan, Eds., ch. 20, CRC Press, 2017, pp. 469–492.
- **CARVALHO, L. O.**, SANTOS, L. F. D., TRAINA, A. J. M., AND TRAINA JR., C. Pivot-Based Similarity Wide-Joins Fostering Near-Duplicate Detection. In *Lecture Notes in Business Information Processing: ICEIS - Revised Selected Papers*, S. Hammoudi, L. Maciaszek, M. M. Missikoff, O. Camp and J. Cordeiro, vol. 291, Eds. Springer, 2017, pp. 81–104.
- **CARVALHO, L. O.**, SANTOS, L. F. D., TRAINA, A. J. M., AND TRAINA JR., C. Quicker than Quick, Meet the QuickNearest: a Fast and Domain-independent Approach to Compute the Similarity k-Nearest Neighbor Join. *Information Systems*, Elsevier, 2018-2019. *Artigo em processo de submissão/revisão.*

Referências Bibliográficas

- [1] ADALI, S., BUFI, C., AND SAPINO, M. L. Ranked relations: query languages and query processing methods for multimedia. *Multimedia Tools and Applications* 24, 3 (Dec. 2004), 197–214.
- [2] ADALI, S., SAPINO, M. L., AND MARSHALL, B. A rank algebra to support multimedia mining applications. In *Proceedings of the 8th International Workshop on Multimedia Data Mining: (associated with the ACM SIGKDD 2007)* (San Jose, CA, USA, 2007), ACM, pp. 5:1–5:9.
- [3] ALY, A. M., AREF, W. G., AND OUZZANI, M. Spatial queries with two knn predicates. *Proceedings of the VLDB Endowment* 5, 11 (July 2012), 1100–1111.
- [4] ARANTES, A. S., VIERA, M. R., TRAINA, A. J. M., AND TRAINA JR., C. The fractal dimension making similarity queries more efficient. In *Proceedings of the 2nd SIGKDD Workshop on Fractals, Power Laws and Other Next Generation Data Mining Tools* (Washington, DC, USA, 2003), ACM, pp. 12–17.
- [5] AYYAZ, S., AND QAMAR, U. Improving collaborative filtering by selecting an effective user neighborhood for recommender systems. In *Proceedings of the 2017 IEEE International Conference on Industrial Technology* (Toronto, ON, Canada, 2017), pp. 1244–1249.
- [6] BALUJA, S., AND COVELL, M. Waveprint: efficient Wavelet-based audio fingerprinting. *Pattern Recognition* 41, 11 (Nov. 2008), 3467–3480.
- [7] BARIONI, M. C. N., KASTER, D. S., RAZENTE, H. L., TRAINA, A. J. M., AND TRAINA JR., C. Querying multimedia data by similarity in relational DBMS. In *Advanced Database Query Systems: Techniques, Applications and Technologies*, L. Yan and Z. Ma, Eds. IGI Global, Hershey, NY, USA, 2011, ch. 14, pp. 323–359.
- [8] BARIONI, M. C. N., RAZENTE, H. L., TRAINA, A. J. M., AND TRAINA JR., C. Seamlessly integrating similarity queries in SQL. *Software - Practice and Experience* 39, 4 (Mar. 2009), 355–384.
- [9] BEDO, M. V. N., TRAINA, A. J. M., AND TRAINA JR. Seamless integration of distance functions and feature vectors for similarity-queries processing. *Journal of Information and Data Management* 5, 3 (Oct. 2014), 308–320.
- [10] BEECKS, C. *Distance-based similarity models for content-based multimedia retrieval*. Dissertation, Rheinisch Westfälische Technische Hochschule Aachen University, Aachen, Germany, July 2013.

- [11] BEECKS, C., UYSAL, M. S., DRIESSEN, P., AND SEIDL, T. Content-based exploration of multimedia databases. In *Proceedings of the 11th International Workshop on Content-Based Multimedia Indexing* (Veszprém, Hungary, 2013), IEEE, pp. 59–64.
- [12] BLYTH, T. S. *Set theory and abstract algebra*, 1st ed. Prentice-Hall, London, UK, 1975.
- [13] BÖHM, C., BRAUNMÜLLER, B., KREBS, F., AND KRIEGEL, H.-P. Epsilon grid order: an algorithm for the similarity join on massive high-dimensional data. *ACM SIGMOD Record* 30, 2 (June 2001), 379–388.
- [14] BÖHM, C., AND KREBS, F. The k-nearest neighbour join: turbo charging the KDD process. *Knowledge and Information Systems* 6, 6 (Nov. 2004), 728–749.
- [15] BRUNO, N. *Automated physical database design and tuning*, 1st ed. CRC Press, Boca Raton, FL, USA, 2011.
- [16] BUDIKOVA, P., BATKO, M., AND ZEZULA, P. Query language for complex similarity queries. In *Proceedings of the 16th East European Conference on Advances in Databases and Information Systems* (Ponzań, Poland, 2012), Springer, pp. 85–98.
- [17] CARVALHO, L. O., OLIVEIRA, W. D., POLA, I. R. V., TRAINA, A. J. M., AND TRAINA JR., C. A wider concept for similarity joins. *Journal of Information and Data Management* 5, 3 (Oct. 2014), 210–223.
- [18] CARVALHO, L. O., SANTOS, L. F. D., OLIVEIRA, W. D., TRAINA, A. J. M., AND TRAINA JR., C. Self similarity wide-joins for near-duplicate image detection. In *Proceedings of the 2015 IEEE International Symposium on Multimedia* (Miami, FL, USA, 2015), Springer, pp. 237–240.
- [19] CARVALHO, L. O., SANTOS, L. F. D., OLIVEIRA, W. D., TRAINA, A. J. M., AND TRAINA JR., C. Similarity joins and beyond: an extended set of binary operators with order. In *Proceedings of the 8th International Conference on Similarity Search and Applications* (Glasgow, UK, 2015), Springer, pp. 29–41.
- [20] CARVALHO, L. O., SANTOS, L. F. D., OLIVEIRA, W. D., TRAINA, A. J. M., AND TRAINA JR., C. Efficient self-similarity range wide-joins fostering near-duplicate image detection in emergency scenarios. In *Proceedings of the 18th International Conference on Enterprise Information Systems* (Rome, Italy, 2016), vol. 1, SciTePress, pp. 81–91.
- [21] CARVALHO, L. O., SANTOS, L. F. D., TRAINA, A. J. M., AND TRAINA JR., C. Pivot-based similarity wide-joins fostering near-duplicate detection. In *Enterprise Information Systems - Revised Selected Papers*, S. Hammoudi, L. A. Maciaszek, M. M. Missikoff, O. Camp, and J. Cordeiro, Eds. Springer, New York, NY, USA, 2017, ch. 4, pp. 81–104.
- [22] CHAUDHURI, S. An overview of query optimization in relational systems. In *Proceedings of the 17th ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems* (Seattle, WA, USA, 1998), ACM, pp. 34–43.
- [23] CHAUDHURI, S., GANTI, V., AND KAUSHIK, R. A primitive operator for similarity joins in data cleaning. In *Proceedings of the 22nd IEEE International Conference on Data Engineering* (USA, 2006), IEEE, pp. 1–12.

- [24] CHÁVEZ, E., NAVARRO, G., BAEZA-YATES, R., AND MARROQUÍN, J. L. Searching in metric spaces. *ACM Computing Surveys* 33, 3 (Sept. 2001), 273–321.
- [25] CHEN, G., YANG, K., CHEN, L., GAO, Y., ZHENG, B., AND CHEN, C. Metric similarity joins using mapreduce. *IEEE Transactions on Knowledge and Data Engineering* 29, 3 (Mar. 2017), 656–669.
- [26] CHEN, L., GAO, Y., LI, X., JENSEN, C. S., AND CHEN, G. Efficient metric indexing for similarity search and similarity joins. *IEEE Transactions on Knowledge and Data Engineering* 29, 3 (Dec. 2017), 556–571.
- [27] CODD, E. F. A relational model of data for large shared data banks. *Communications of the ACM* 13, 6 (June 1970), 377–387.
- [28] DATE, C. J. *SQL and relational theory: how to write accurate SQL code*, 2nd ed. O'Reilly Media, Sebastopol, CA, USA, 2011.
- [29] DATTA, R., JOSHI, D., LI, J., AND WANG, J. Z. Image retrieval: ideas, influences, and trends of the new age. *ACM Computing Surveys* 40, 2 (Apr. 2008), 1–60.
- [30] DEZA, M. M., AND DEZA, E. *Encyclopedia of distances*, 2nd ed. Springer, New York, NY, USA, 2013.
- [31] DOHNAL, V., GENNARO, C., AND ZEZULA, P. Similarity join in metric spaces using eD-index. In *Proceedings of the 14th International Conference on Database and Expert Systems Applications* (Prague, Czech Republic, 2003), pp. 484–493.
- [32] ELMASRI, R., AND NAVATHE, S. *Fundamentals of database systems*, 6th ed. Addison-Wesley, Boston, MA, USA, 2011.
- [33] FALOUTSOS, C. *Searching multimedia database by content*, 1st ed. Kluwer Academic Publisher, Norwell, MA, USA, 1996.
- [34] FERREIRA, M. R. P., SANTOS, L. F. D., TRAINA, A. J. M., DIAS, I., CHBEIR, R., AND TRAINA JR., C. Algebraic properties to optimize kNN queries. *Journal of Information and Data Management* 2, 3 (Oct. 2011), 385–400.
- [35] FERREIRA, M. R. P., TRAINA, A. J. M., DIAS, I., CHBEIR, R., AND TRAINA JR., C. Identifying algebraic properties to support optimization of unary similarity queries. In *Proceedings of the 3rd Alberto Mendelzon International Workshop on Foundations of Data Management* (Arequipa, Peru, 2009), CEUR-WS, pp. 1–10.
- [36] FREDRIKSSON, K., AND BRAITHWAITE, B. Quicker similarity joins in metric spaces. In *Proceedings of the 6th International Conference on Similarity Search and Applications* (A Coruña, Spain, 2013), Springer, pp. 127–140.
- [37] FREDRIKSSON, K., AND BRAITHWAITE, B. Quicker range- and k-nn joins in metric spaces. *Information Systems* 52, C (Aug. 2015), 189–204.
- [38] GAO, Y., CHEN, L., LI, X., YAO, B., AND CHEN, G. Efficient k-closest pair queries in general metric spaces. *VLDB Journal* 24, 3 (Apr. 2015), 415–439.

- [39] GARCIA-MOLINA, H., ULLMAN, J. D., AND WIDOM, J. *Database system implementation*, 1st ed. Prentice-Hall, Upper Saddle River, NJ, USA, 2000.
- [40] GRAEFE, G. Query evaluation techniques for large databases. *ACM Computing Surveys* 25, 2 (June 1993), 73–169.
- [41] GRAEFE, G. New algorithms for join and grouping operations. *Computer Science - Research and Development* 27, 1 (Feb. 2012), 3–27.
- [42] GRAFEN, P. W. P. J., AND BY, R. A. A multi-set extended relational algebra: a formal approach to a practical issue. In *Proceedings of the 10th International Conference on Data Engineering* (Houston, TX, USA, 1994), IEEE, pp. 80–88.
- [43] GULIATO, D., MELO, E. V., RANGAYAN, R. M., AND SOARES, R. C. POSTGRESQL-IE: an image-handling extension for PostgreSQL. *Journal of Digital Imaging* 22, 2 (Apr. 2009), 149–165.
- [44] HETLAND, M. L. The basic principles of metric indexing. In *Swarm Intelligence for Multi-Objective Problems in Data Mining*, C. A. Coello Coello, S. Dehuri, and S. Ghosh, Eds. Springer, Chennai, India, 2009, ch. 9, pp. 199–232.
- [45] HJALTASON, G. R., AND SAMET, H. Incremental distance join algorithms for spatial databases. *ACM SIGMOD Record* 27, 2 (June 1998), 237–248.
- [46] HJALTASON, G. R., AND SAMET, H. Index-driven similarity search in metric spaces. *ACM Transactions on Database Systems* 28, 4 (Dec. 2003), 517–580.
- [47] HU, Y., YANG, C., JI, C., XU, Y., AND LI, X. Efficient snapshot knn join processing for large data using mapreduce. In *Proceedings of the 22nd International Conference on Parallel and Distributed Systems* (Wuhan, China, 2016), IEEE, pp. 713–720.
- [48] JACOX, E. H., AND SAMET, H. Iterative spatial join. *ACM Transactions on Database Systems* 28, 3 (Sept. 2003), 230–256.
- [49] JACOX, E. H., AND SAMET, H. Metric space similarity joins. *ACM Transactions on Database Systems* 33, 2 (June 2008), 7:1–7:38.
- [50] KASTER, D. S., BUGATTI, P. H., SILVA, M. P., TRAINA, A. J. M., MARQUES, P. M. A., SANTOS, A. C., AND TRAINA JR., C. MedFMI-SiR: a powerful DBMS solution for large-scale medical image retrieval. In *Proceedings of the 2nd International Conference on Information Technology in Bio and Medical Informatics* (Toulouse, France, 2011), Springer, pp. 16–30.
- [51] KASUTANI, E., AND YAMADA, A. The mpeg-7 color layout descriptor: a compact image feature description for high-speed image/video segment retrieval. In *Proceedings of the 8th International Conference on Image Processing* (Thessaloniki, Greece, 2001), IEEE, pp. 674–667.
- [52] LEVENSHTAIN, V. I. Binary codes capable of correcting deletions, insertions, and reversals. *Soviet Physics Doklady* 10, 8 (Feb. 1966), 707–710.
- [53] LI, C., CHANG, K. C.-C., ILYAS, I. F., AND SONG, S. RankSQL: query algebra and optimization for relational top-k queries. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Baltimore, MD, USA, 2005), ACM, pp. 131–142.

- [54] LI, C., WANG, M., LIM, L., WANG, H., AND CHANG, K. C.-C. Supporting ranking and clustering as generalized order-by and group-by. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Beijing, China, 2007), ACM, pp. 127–138.
- [55] LI, G., DENG, D., WANG, J., AND FENG, J. Pass-join: a partition-based method for similarity joins. *Proceedings of the VLDB Endowment* 5, 3 (Nov. 2011), 253–264.
- [56] LI, J., QIAN, X., LI, Q., ZHAO, Y., WANG, L., AND TANG, Y. Y. Mining near duplicate image groups. *Multimedia Tools and Applications* 74, 2 (Jan. 2015), 655–669.
- [57] LIMA, E. L. *Espaços métricos*, 5th ed. Springer, Rio de Janeiro, RJ, Brasil, 2011.
- [58] LO, M. L., AND RAVISHANKAR, C. V. Spatial hash-joins. *ACM SIGMOD Record* 25, 2 (June 1996), 247–258.
- [59] LU, W., HOU, J., YAN, Y., ZHANG, M., DU, X., AND MOSCIBRODA, T. Msql: efficient similarity search in metric spaces using sql. *The VLDB Journal* 26, 6 (Dec. 2017), 829–854.
- [60] MANOLOPOULOS, Y., NANOPOULOS, A., PAPADOPOULOS, A. N., AND THEODORIDIS, Y. *R-tress: theory and applications*, 1st ed. Springer, London, UK, 2006.
- [61] MARRI, W. J. A., MALLUHI, Q. M., OUZZANI, M., TANG, M., AND AREF, W. G. The similarity-aware relational database set operators. *Information Systems* 59 (July 2016), 79–93.
- [62] McCLAIN, J., AND KUMAR, P. Fast k-clustering queries on embeddings of road networks. In *Proceedings of the 3rd International Conference on Computing for Geospatial Research and Applications* (2012), ACM, pp. 11:1–11:9.
- [63] MELTON, J., AND SIMON, A. R. *SQL:1999: understanding relational language components*, 1st ed. Morgan Kaufmann, San Francisco, CA, USA, 2001.
- [64] MISHRA, P., AND EICH, M. H. Join processing in relational databases. *ACM Computing Surveys* 24, 1 (Mar. 1992), 63–113.
- [65] MORALES, G. F., AND GIONIS, A. Streaming similarity self-join. *Proceedings of the VLDB Endowment* 9, 10 (June 2016), 792–803.
- [66] OLIVEIRA, P. H., FRAIDEINBERZE, A. C., LAVERDE, N. A., COLMENARES, H. A. G., GONZAGA, A. S., FERREIRA, L. D., OLIVEIRA, W. D., RODRIGUES JR., J. F., CORDEIRO, R. L. F., TRAINA JR., C., TRAINA, A. J. M., AND SOUSA, E. P. M. On the support of a similarity-enabled relational database management system in civilian crisis situations. In *Proceedings of the 18th International Conference on Enterprise Information Systems* (Rome, Italy, 2016), vol. 1, SciTePress, pp. 119–126.
- [67] PAREDES, R., AND REYES, N. Solving similarity joins and range queries in metric spaces with the list of twin clusters. *Journal of Discrete Algorithms* 7, 1 (Mar. 2009), 18–35.
- [68] QIN, J., ZHOU, X., WANG, W., AND XIAO, C. Trie-based similarity search and join. In *Joint Extending Database Technology and Database Theory International Conferences* (Genoa, Italy, 2013), pp. 392–396.

- [69] RONG, C., SILVA, Y. N., AND LI, C. String similarity join with different similarity thresholds based on novel indexing techniques. *Frontiers of Computer Science* 11, 2 (Apr. 2017), 307–319.
- [70] SANTOS, L. F. D., CARVALHO, L. O., OLIVEIRA, W. D., TRAINA, A. J. M., AND TRAINA JR., C. Diversity in similarity joins. In *Proceedings of the 8th International Conference on Similarity Search and Applications* (Glasgow, UK, 2015), Springer, pp. 42–53.
- [71] SILBERSCHATZ, A., KORTH, H. F., AND SUDARSHAN, S. *Database system concepts*, 6th ed. McGraw-Hill, New York, NY, USA, 2010.
- [72] SILVA, Y. N., ALY, A. M., AREF, W. G., AND LARSON, P.-A. SimDB: a similarity-aware database system. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (Indianapolis, IN, USA, 2010), ACM, pp. 1243–1246.
- [73] SILVA, Y. N., AREF, W. G., AND ALI, M. H. Similarity group-by. In *Proceedings of the 25th IEEE International Conference on Data Engineering* (Shanghai, China, 2009), IEEE Computer Society, pp. 904–915.
- [74] SILVA, Y. N., AREF, W. G., AND ALI, M. H. The similarity join database operator. In *Proceedings of the 26th IEEE International Conference on Data Engineering* (Long Beach, CA, USA, 2010), IEEE Computer Society, pp. 892–903.
- [75] SILVA, Y. N., AREF, W. G., LARSON, P.-A., PEARSON, S., AND ALI, M. H. Similarity queries: their conceptual evaluation, transformations, and processing. *The VLDB Journal* 22, 3 (June 2013), 395–420.
- [76] SILVA, Y. N., PEARSON, S., AND CHENEY, J. A. Database similarity join for metric spaces. In *Proceedings of the 6th International Conference on Similarity Search and Applications* (La Coruña, Spain, 2013), Springer, pp. 266–279.
- [77] SILVA, Y. N., PEARSON, S. S., CHON, J., AND ROBERTS, R. Similarity joins: Their implementation and interactions with other database operators. *Information Systems* 52 (Aug. 2015), 149–162.
- [78] SOUSA, E. P. M., TRAINA JR., C., TRAINA, A. J. M., WU, L., AND FALOUTSOS, C. A fast and effective method to find correlations among attributes in databases. *Data Mining and Knowledge Discovery* 14, 3 (June 2007), 367–407.
- [79] STRICKER, M., AND ORENGO, M. Similarity of color images. In *Proceedings of the 3rd SPIE Conference on Storage and Retrieval for Image and Video Databases* (San Jose, CA, USA, 1995), vol. 2420, pp. 381–392.
- [80] TALAVERA-LLAMES, R. L., PÉREZ-CHACÓN, R., MARTÍNEZ-BALLESTEROS, M., TRONCOSO, A., AND MARTÍNEZ-ÁLVAREZ, F. A nearest neighbours-based algorithm for big time series data forecasting. In *Proceedings of the 11th International Conference on Hybrid Artificial Intelligent Systems* (Seville, Spain, 2016), pp. 174–185.
- [81] TRAINA JR., C., TRAINA, A. J. M., SEEGER, B., AND FALOUTSOS, C. Slim-trees: High performance metric trees minimizing overlap between nodes. In *Proceedings of the 7th International Conference on Extending Database Technology: Advances in Database Technology* (Konstanz, Germany, 2000), pp. 51–65.

- [82] TRAINA JR., C., TRAINA, A. J. M., VIEIRA, M. R., ARANTES, A. S., AND FALOUTSOS, C. Efficient processing of complex similarity queries in rdbms through query rewriting. In *Proceedings of the 15th ACM International Conference on Information and Knowledge Management* (Arlington, VA, USA, 2006), ACM, pp. 4–13.
- [83] WANG, J., LI, G., AND FENG, J. Extending string similarity join to tolerant fuzzy token matching. *ACM Transactions on Database Systems* 39, 1 (Jan. 2014), 7:1–7:45.
- [84] XIAO, C., WANG, W., LIN, X., AND SHANG, H. Top-k set similarity joins. In *Proceedings of the 25th IEEE International Conference on Data Engineering* (Shanghai, China, 2009), IEEE Computer Society, pp. 916–927.
- [85] XIAO, C., WANG, W., LIN, X., YU, J. X., AND WANG, G. Efficient similarity joins for near-duplicate detection. *ACM Transactions on Database Systems* 36, 3 (Aug. 2011), 15:1–15:41.
- [86] YANG, C., YU, X., AND LIU, Y. Continuous knn join processing for real-time recommendation. In *Proceedings of the 2014 IEEE International Conference on Data Mining* (Shenzhen, China, 2014), pp. 640–649.
- [87] YANG, C., YU, X., AND LIU, Y. Towards efficient knn joins on data streams. In *Proceedings of the 2014 IEEE International Congress on Big Data* (Anchorage, AK, USA, 2014), pp. 782–783.
- [88] YU, C., CUI, B., WANG, S., AND SU, J. Efficient index-based kNN join processing for high-dimensional data. *Information and Software Technology* 49, 4 (Apr. 2007), 332–344.
- [89] YU, C., ZHANG, R., HUANG, Y., AND XIONG, H. High-dimensional knn joins with incremental updates. *GeoInformatica* 14, 1 (Jan. 2010), 55–82.
- [90] YU, M., LI, G., DENG, D., AND FENG, J. String similarity search and join: a survey. *Frontiers of Computer Science* 10, 3 (June 2016), 399–417.
- [91] YUAN, P., SHA, C., AND SUN, Y. Hash ed-join: Approximate string similarity join with hashing. In *Proceedings of the 19th International Conference on Database Systems for Advanced Applications* (Bali, Indonesia, 2014), Springer, pp. 217–229.
- [92] ZEZULA, P. Multi feature indexing network MUFIN for similarity search applications. In *Proceedings of the 38th International Conference on Current Trends in Theory and Practice of Computer Science* (Špindlerův Mlýn, Czech Republic, 2012), Springer, pp. 77–87.
- [93] ZEZULA, P., AMATO, G., DOHNAL, V., AND BATKO, M. *Similarity search: the metric space approach*, 1st ed. Springer, New York, NY, USA, 2006.
- [94] ZHANG, C., LI, F., AND JESTES, J. Efficient parallel kNN joins for large data in MapReduce. In *Proceedings of the 15th International Conference on Extending Database Technology* (Berlin, Germany, 2012), ACM, pp. 38–49.
- [95] ZHANG, D., CHAN, C.-Y., AND TAN, K.-L. Nearest group queries. In *Proceedings of the 25th International Conference on Scientific and Statistical Database Management* (Baltimore, MD, USA, 2013), ACM, pp. 7:1–7:12.

Distância para um hiperplano generalizado

Embora um espaço n -dimensional permita computar com precisão a distância de um elemento arbitrário a um hiperplano gerado por dois pivôs, esta característica não se faz presente em espaços métricos genéricos.

Espaços n -dimensionais são importantes instâncias de espaços métricos, permitindo definir hiperplanos “reais” e computar a distância exata de qualquer elemento ao hiperplano usando funções de distância como, por exemplo, a L_p . Espaços métricos genéricos, no entanto, não garantem a existência de um hiperplano “real”, uma vez que vários tipos de dados não são representados como pontos n -dimensionais. Por exemplo, dados representados como cadeias de caracteres e comparados através da distância de edição (e.g., palavras de dicionário, proteínas), ou conjuntos de elementos que empregam métricas como a distância de Jaccard não mantêm essa propriedade.

Dados puramente métricos não podem ser representados em espaços n -dimensionais e, portanto, não é possível traçar um hiperplano de separação entre eles. O hiperplano é dito ser “generalizado”, i.e., embora não seja possível representá-lo, ainda é possível efetuar o particionamento dos elementos usando somente uma função de distância. Nestes casos, a distância real para o hiperplano não pode ser diretamente determinada (porque o hiperplano não existe de fato), mas é possível estabelecer um limite inferior para esta distância, que é dado pela equação 3.1.

Este limite inferior expresso pela equação 3.1 é baseado no seguinte Lema 1 [46].

Lema 1. *Seja pA e pB elementos de referência e h um elemento que não esteja mais próximo de pB que de pA , i.e., $d(h, pA) \leq d(h, pB)$ (figura A.1). Dados um elemento de consulta q e as distâncias $d(q, pA)$ e $d(q, pB)$, um limite inferior para o valor $d(q, h)$ é dado pela equação A.1.*

$$\max\left(\frac{d(q, pA) - d(q, pB)}{2}, 0\right) \leq d(q, h) \quad (\text{A.1})$$

Demonstração. Pela desigualdade triangular: $d(q, pA) \leq d(q, h) + d(h, pA)$.

Rearranjando os termos, é possível obter as seguintes expressões A.2 e A.3.

Figura A.1: Limite inferior da distância de um objeto ao hiperplano

$$d(q, pA) - d(q, h) \leq d(h, pA) \quad (\text{A.2})$$

$$d(h, pB) \leq d(q, pB) + d(q, h) \quad (\text{A.3})$$

Como o Lema 1 assume que $d(h, pA) \leq d(h, pB)$, a partir das equações A.2 and A.3 é possível realizar a substituição dos membros da propriedade assumida no Lema e obter a equação A.4.

$$d(q, pA) - d(q, h) \leq d(q, pB) - d(q, h) \quad (\text{A.4})$$

Rearranjando os termos da equação A.4, obtém-se $d(q, pA) - d(q, pB) \leq 2 \cdot d(q, h)$, que é o primeiro parâmetro da função *max*. O segundo parâmetro, que é o valor zero, garante que a propriedade da não-negatividade se mantenha. $\square$

No algoritmo 3.7, contudo, é permitido à distância computada pela equação 3.1 ser negativa (o que ocorre para elementos mais próximos a pB que de pA). Entretanto, considera-se o valor absoluto desta distância, o que justifica o fator -1 no passo 9 do algoritmo.

Demonstração das propriedades algébricas envolvendo wide-joins

Propriedade 1. *Comutatividade de range wide-join.*

$$\underbrace{R \underset{\bowtie}{\text{Rng}(d, \xi)} S}_{(i)} = \underbrace{S \underset{\bowtie}{\text{Rng}(d, \xi)} R}_{(ii)}$$

Demonstração. Segue diretamente da simetria de d :

$$\begin{aligned} (i) &= \{t' \in \{t[r, s, \text{ord}] : t \in (R \times S) \mid d(t.A_1, t.A_2) \leq \xi \mid t'.\text{ord} \leq \kappa_f\} \\ &= \{t' \in \{t[s, r, \text{ord}] : t \in (S \times R) \mid d(t.A_2, t.A_1) \leq \xi \mid t'.\text{ord} \leq \kappa_f\} \\ &= (ii) \end{aligned}$$

□

Propriedade 2 (Distributividade de projeção sobre wide-join).

$$\underbrace{\pi_{\{r_i, \dots, A_1, s_j, \dots, A_2\}} \left(R \underset{\bowtie}{\text{Rng}(d, \xi)} S \right)}_{(i)} = \underbrace{\left(\pi_{\{r_i, A_1\}}(R) \right) \underset{\bowtie}{\text{Rng}(d, \xi)} \left(\pi_{\{s_j, A_2\}}(S) \right)}_{(ii)}$$

Demonstração.

$$\begin{aligned} (i) &= \pi_{\{r_i, \dots, A_1, s_j, \dots, A_2\}} \{t' \in \{t[r, s, \text{ord}] : t \in (R \times S) \mid t.A_1 \theta_s t.A_2 \mid t'.\text{ord} \leq \kappa_f\} \\ &= \{t''[r_i, A_1, s_j, A_2] : t' \in \{t' \in \{t[r, s, \text{ord}] : t \in (R \times S) \mid t.A_1 \theta_s t.A_2 \mid t'.\text{ord} \leq \kappa_f\}\} \\ &= \{t'[r_i, A_1, s_j, A_2] \mid t' \in \{t[r, s, \text{ord}] : t \in (R \times S) \mid t.A_1 \theta_s t.A_2 \wedge t'.\text{ord} \leq \kappa_f\} \\ &= \{t'' \in \{t'[r, s, \text{ord}] : t' \in \{(t[r_i, A_1] : t \in R) \times (t[s_j, A_2] : t \in S)\} \mid t'.A_1 \theta_s t'.A_2 \mid t''.\text{ord} \leq \kappa_f\} \\ &= \{t' \in \{t[r, s, \text{ord}] : t \in \{(\pi_{\{r_i, A_1\}}(R)) \times (\pi_{\{s_j, A_2\}}(S))\} \mid t.A_1 \theta_s t.A_2 \mid t'.\text{ord} \leq \kappa_f\} \\ &= (ii) \end{aligned}$$

□

Propriedade 3 (Distributividade de *wide-join* sobre união).

$$\underbrace{T \overset{(A_3 \theta_s A_4), \kappa_f}{\bowtie} (R \cup S)}_{(i)} \subseteq \underbrace{\left(T \overset{(A_3 \theta_s A_1), \kappa_f}{\bowtie} R \right) \cup \left(T \overset{(A_3 \theta_s A_2), \kappa_f}{\bowtie} S \right)}_{(ii)} \underbrace{\hspace{10em}}_{ii.a \quad ii.b}$$

Demonstração. Primeiramente, assume-se que as relações envolvidas nesta propriedade possuem esquemas compatíveis em união. Assim:

(i) = $T \overset{(A_3 \theta_s A_4), \kappa_f}{\bowtie} (R \cup S) = \{a_1, a_2, \dots, a_p\}, 0 < p \leq \kappa_f$, onde:

$$a_1 = \{t' \in \{t[r_1, \dots, r_n, s_1, \dots, s_m, ord] : t \in (T \times U) \mid t.A_3 \theta_s t.A_4 \mid t'.ord \leq \kappa_f\}$$

$$a_2 = \{t' \in \{t[r_1, \dots, r_n, s_1, \dots, s_m, ord] : t \in (T \times U) - \{a_1\} \mid t.A_3 \theta_s t.A_4 \mid t'.ord \leq \kappa_f\}$$

$$a_3 = \{t' \in \{t[r_1, \dots, r_n, s_1, \dots, s_m, ord] : t \in (T \times U) - \{a_1, a_2\} \mid t.A_3 \theta_s t.A_4 \mid t'.ord \leq \kappa_f\}$$

$$a_4 = \{t' \in \{t[r_1, \dots, r_n, s_1, \dots, s_m, ord] : t \in (T \times U) - \{a_1, a_2, a_3\} \mid t.A_3 \theta_s t.A_4 \mid t'.ord \leq \kappa_f\}$$

⋮

$$a_p = \{t' \in \{t[r_1, \dots, r_n, s_1, \dots, s_m, ord] : t \in (T \times U) - \{a_1, \dots, a_{p-1}\} \mid t.A_3 \theta_s t.A_4 \mid t'.ord \leq \kappa_f\}$$

sendo $U = R \cup S$, $A_4 \subseteq U$ um dos atributos complexos $A_1 \subset R$ ou $A_2 \subset S$ e

$$(t.A_3 \theta_s t.A_4 \wedge t.ord \leq \kappa_f) \Rightarrow t \in (T \times U) \mid \forall t' \in (T \times U) : d(t.A_3, t.A_4) \leq d(t'.A_3, t'.A_4).$$

Assim, da definição 17:

$$a_1 = \{a \in (T \times U) \mid \forall a' \in (T \times U) : d(a.A_3, a.A_4) \leq d(a'.A_3, a'.A_4)\}$$

$$a_2 = \{a \in (T \times U) \mid \forall a' \in (T \times U) - \{a_1\} : d(a.A_3, a.A_4) \leq d(a'.A_3, a'.A_4)\}$$

$$a_3 = \{a \in (T \times U) \mid \forall a' \in (T \times U) - \{a_1, a_2\} : d(a.A_3, a.A_4) \leq d(a'.A_3, a'.A_4)\}$$

$$a_4 = \{a \in (T \times U) \mid \forall a' \in (T \times U) - \{a_1, a_2, a_3\} : d(a.A_3, a.A_4) \leq d(a'.A_3, a'.A_4)\}$$

⋮

$$a_p = \{a \in (T \times U) \mid \forall a' \in (T \times U) - \{a_1, \dots, a_{p-1}\} : d(a.A_3, a.A_4) \leq d(a'.A_3, a'.A_4)\}$$

De modo análogo:

$$(ii.a) = T \overset{(A_3 \theta_s A_1), \kappa_f}{\bowtie} R = \{b_1, \dots, b_p\}, 0 < p \leq \kappa_f, \text{ onde:}$$

$$\begin{aligned}
b_1 &= \{b \in (T \times R) \mid \forall t' \in (T \times R) : d(b.A_3, b.A_1) \leq d(b'.A_3, b'.A_1)\} \\
&\vdots \\
b_p &= \{b \in (T \times R) - \{b_1, \dots, b_{p-1}\} \mid \forall b' \in (T \times R) - \{b_1, \dots, b_{p-1}\} : d(b.A_3, b.A_1) \leq d(b'.A_3, b'.A_1)\}
\end{aligned}$$

e,

$$(ii.b) = T \stackrel{(A_3 \theta_s A_2), \kappa_f}{\bowtie} S = \{c_1, \dots, c_p\}, 0 < p \leq \kappa_f, \text{ onde:}$$

$$\begin{aligned}
c_1 &= \{c \in (T \times S) \mid \forall c' \in (T \times S) : d(c.A_3, c.A_2) \leq d(c'.A_3, c'.A_2)\} \\
&\vdots \\
c_p &= \{c \in (T \times S) - \{c_1, \dots, c_{p-1}\} \mid \forall c' \in (T \times S) - \{c_1, \dots, c_{p-1}\} : d(c.A_3, c.A_2) \leq d(c'.A_3, c'.A_2)\}
\end{aligned}$$

Induzindo sobre p :

Base da indução: para $p = 1$, tem-se que $\{a_1\} \subseteq \{b_1\} \cup \{c_1\}$. Como

$$a_1 \in T \times (R \cup S)$$

$$b_1 \in T \times R$$

$$c_1 \in T \times S$$

permite enunciar que:

$$\begin{aligned}
d(a_1.A_3, a_1.A_4) &= \min(d(a.A_3, a.A_4), a \in T \times (R \cup S)) \\
&\leq \min(d(b.A_3, b.A_1), b \in T \times R), \\
&\leq \min(d(c.A_3, c.A_2), c \in T \times S)
\end{aligned}$$

Portanto:

$$\begin{aligned}
d(a_1.A_3, a_1.A_4) &\leq \min(\min(d(b.A_3, b.A_1), b \in T \times R), \min(d(c.A_3, c.A_2), c \in T \times S)) \\
&= \min(d(b_1.A_3, b_1.A_1), d(c_1.A_3, c_1.A_2))
\end{aligned}$$

Ou seja: $a_1 \in \{b_1, c_1\}$.

Como a_p pertence a (i), b_p pertence a (ii.a) e c_p pertence a (ii.b):

$$\{a_p\} \subseteq \{b_p, c_p\} = \{a_p\} \subseteq \{b_p\} \cup \{c_p\}, \text{ e a propriedade 3 é verdadeira para } p = 1.$$

Fazendo $p = 2$:

$a_2 \in T \times (R \cup S)$, o que permite enunciar que:

$$\begin{aligned}
d(a_2.A_3, a_2.A_4) &= \min(d(a.A_3, a.A_4), a \in T \times (R \cup S) - \{a_1\}) \\
&\leq \min(d(b.A_3, b.A_1), b \in T \times R - \{a_1\}), \\
&\leq \min(d(c.A_3, c.A_2), c \in T \times S - \{a_1\})
\end{aligned}$$

Portanto:

$$d(a_2.A_3, a_2.A_4) \leq \min(\min(d(b.A_3, b.A_1), b \in T \times R - \{a_1\}), \min(d(c.A_3, c.A_2), c \in T \times S - \{a_1\}))$$

Como $a_1 \in \{b_1, c_1\}$, se $a_1 = b_1 = c_1$, então:

$$d(a_2.A_3, a_2.A_4) \leq \min(d(b_2.A_3, b_2.A_1), d(c_2.A_3, c_2.A_2)) \quad (\text{B.1})$$

e $a_2 \in \{b_2, c_2\}$, e logo $\{a_1, a_2\} \subseteq \{b_1, b_2\} \cup \{c_1, c_2\}$ (lembrando que $b_1 = c_1$).

Entretanto, se $a_1 = b_1$ e $a_1 \neq c_1$, então:

$$\begin{aligned}
d(a_2.A_3, a_2.A_4) &\leq \min(d(b_2.A_3, b_2.A_1), \min(d(c.A_3, c.A_2), c \in T \times S - \{a_1\})) \\
&\leq \min(d(b_2.A_3, b_2.A_1), d(c_1.A_3, c_1.A_2))
\end{aligned} \quad (\text{B.2})$$

e $a_2 \in \{b_2, c_1\}$, e logo $\{a_1, a_2\} \subseteq \{b_1, b_2\} \cup \{c_1\}$.

Entretanto, supondo agora que $a_1 \neq b_1$ e $a_1 = c_1$, então:

$$\begin{aligned}
d(a_2.A_3, a_2.A_4) &\leq \min(\min(d(b.A_3, b.A_1), b \in T \times R - \{a_1\}), d(c_2.A_3, c_2.A_2)) \\
&\leq \min(d(b_1.A_3, b_1.A_1), d(c_2.A_3, c_2.A_2))
\end{aligned} \quad (\text{B.3})$$

e $a_2 \in \{b_1, c_2\}$, e logo $\{a_1, a_2\} \subseteq \{b_1\} \cup \{c_1, c_2\}$.

Portanto, das três equações B.1, B.2 e B.2 anteriores: $\{a_1, a_2\} \subseteq \{b_1, b_2\} \cup \{c_1, c_2\}$, e a propriedade 3 é verdadeira para $p = 2$.

Hipótese de indução: assumindo a propriedade 3 para um inteiro “ i ” arbitrário, tem-se:

$$\{a_1, \dots, a_i\} \subseteq \{b_1, \dots, b_i\} \cup \{c_1, \dots, c_i\}$$

Ou seja, é verdade que:

$$\begin{aligned} d(a_i.A_3, a_i.A_4) &= \min(d(a.A_3, a.A_4), a \in T \times (R \cup S) - \{a_1, \dots, a_{i-1}\}) \\ &\leq \min(d(b.A_3, b.A_1), b \in T \times R - \{a_1, \dots, a_{i-1}\}), \\ &\leq \min(d(c.A_3, c.A_2), c \in T \times S - \{a_1, \dots, a_{i-1}\}) \end{aligned}$$

Agora, o passo indutivo: $p = i + 1$.

$$\{a_1, \dots, a_{i+1}\} \stackrel{?}{\subseteq} \{b_1, \dots, b_{i+1}\} \cup \{c_1, \dots, c_{i+1}\}$$

Pela hipótese de indução, $a_i \in \{b_i, c_i\}$ e, portanto:

$$\begin{aligned} d(a_{i+1}.A_3, a_{i+1}.A_4) &\stackrel{?}{\leq} \min(d(a.A_3, a.A_4), a \in T \times (R \cup S) - \{a_1, \dots, a_i\}) \\ &\stackrel{?}{\leq} \min(d(b.A_3, b.A_1), b \in T \times R - \{a_1, \dots, a_i\}), \\ &\stackrel{?}{\leq} \min(d(c.A_3, c.A_2), c \in T \times S - \{a_1, \dots, a_i\}) \end{aligned}$$

Se $a_i = b_i = c_i$, então:

$$d(a_{i+1}.A_3, a_{i+1}.A_4) \leq \min(d(b_{i+1}.A_3, b_{i+1}.A_1), d(c_{i+1}.A_3, c_{i+1}.A_2)) \quad (\text{B.4})$$

e $a_{i+1} \in \{b_{i+1}, c_{i+1}\}$, e logo $\{a_1, \dots, a_i, a_{i+1}\} \subseteq \{b_1, \dots, b_i, b_{i+1}\} \cup \{c_1, \dots, c_i, c_{i+1}\}$ (lembrando que $b_i = c_i$).

Para $a_i = b_i$ e $a_i \neq c_i$, então:

$$\begin{aligned} d(a_{i+1}.A_3, a_{i+1}.A_4) &\leq \min(d(b_{i+1}.A_3, b_{i+1}.A_1), \min(d(c.A_3, c.A_2), c \in T \times S - \{a_i, \dots, a_i\})) \\ &\leq \min(d(b_{i+1}.A_3, b_{i+1}.A_1), d(c_j.A_3, c_j.A_2)), j \in [1, i] \end{aligned} \quad (\text{B.5})$$

e $a_{i+1} \in \{b_{i+1}, c_j\}$, e logo $\{a_1, \dots, a_i, a_{i+1}\} \subseteq \{b_1, \dots, b_{i-1}, b_i, b_{i+1}\} \cup \{c_1, \dots, c_{i-1}, c_i\}$.

Entretanto, se $a_i \neq b_i$ e $a_i = c_i$, então:

$$\begin{aligned} d(a_{i+1}.A_3, a_{i+1}.A_4) &\leq \min(\min(d(b.A_3, b.A_1), b \in T \times R - \{a_1, \dots, a_i\}), d(c_{i+1}.A_3, c_{i+1}.A_2)) \\ &\leq \min(d(b_j.A_3, b_j.A_1), d(c_{i+1}.A_3, c_{i+1}.A_2)), j \in [1, i] \end{aligned} \quad (\text{B.6})$$

e $a_{i+1} \in \{b_j, c_{i+1}\}$, e logo $\{a_1, \dots, a_i, a_{i+1}\} \subseteq \{b_1, \dots, b_{i-1}, b_i\} \cup \{c_1, \dots, c_{i-1}, c_i, c_{i+1}\}$.

Portanto, com base nas equações B.4, B.5 e B.6:

$$\{a_1, \dots, a_{i-1}, a_i, a_{i+1}\} \subseteq \{b_1, \dots, b_{i-1}, b_i, b_{i+1}\} \cup \{c_1, \dots, c_{i-1}, c_i, c_{i+1}\}$$

e a propriedade 3 é verdadeira para $p = i + 1$, confirmando a hipótese de indução. $\square$