Aprimoramentos da Junção Canalizada aplicada em dados Métricos e Espaciais

Renzo Paranaíba Mesquita

Aprimoramentos da Junção Canalizada aplicada em dados Métricos e Espaciais

Renzo Paranaíba Mesquita

Orientador: *Prof. Dr. Enzo Seraphim*

Dissertação submetida ao Programa de Pós-Graduação em Ciência e Tecnologia da Computação como parte dos requisitos para obtenção do título de Mestre em Ciência e Tecnologia da Computação.

UNIFEI - Itajubá - MG
Junho de 2017

Aos meus pais, com todo meu amor e gratidão.

“Revelemo-nos mais por atos do que por palavras.”
Theodomiro Carneiro Santiago

Agradecimentos

Com lágrimas nos olhos me sentei para escrever esta seção. Depois de tantos desafios, nunca pensei que este momento poderia chegar.

O escopo desta jornada foi grande, regado de classes e objetos desafiadores, mas que no final, renderam experiências e conhecimentos que jamais imaginei adquirir. Felizmente, no decorrer deste desafio e bem no meio deste escopo, tive o privilégio de ter o apoio de uma classe bastante especial chamada Anjo, especializada em três outras classes que foram responsáveis por instanciarem diversos objetos que, de forma direta ou indireta, me ajudaram a cumprir esta jornada. Estas subclasses são: Família, Amigo e Mestre.

Começo agradecendo a alguns dos objetos pertencentes à classe Família, responsável por dar origem aos meus objetos mais preciosos, que juntos, formam a minha base de sustentação. Pai, meu maior exemplo de força física, que mesmo com suas restrições decorrentes de um infarto, nunca sequer demonstrou fraqueza ou poupou esforços para me levantar em momentos de dificuldade. Mãe, meu maior exemplo de força psicológica e meu objeto mais precioso. Com sua sabedoria e inteligência ímpar, por incontáveis vezes me fortaleceu com palavras e lições de conforto. Minhã irmã, obrigado por inúmeras vezes ter me acolhido em sua residência nesta jornada e por estar sempre me apoiando em minhas realizações.

Agradeço também a objetos especiais pertencentes à classe Amigo, responsável por dar origem aos anjos que estiveram presentes em diversos momentos pelo qual precisei de suporte e compreensão. Ao amigo e agora mestre, Diógenes Leonel, meu muito obrigado por estar presente no começo desta jornada. Jamais me esquecerei dos momentos de estudos intensos mas também de descontração que vivemos juntos. Pode ter certeza que a nossa amizade deixou esta jornada bem menos dolorosa. Obrigado Debi, por também ter me dado suporte no início desta jornada, e Duda, por apoiar e me confortar no final deste trabalho que é muito importante pra mim.

Por fim, aproveito também para exaltar objetos pertencentes à terceira e última subclasse de Anjo, denominada Mestre, responsável por dar origem aos meus objetos mais inspiradores, que sem eles, este trabalho seria apenas um sonho. Começo agradecendo meu grande colega de trabalho, Prof. Me. João Paulo Carvalho Henriques, pelo compartilhamento de dicas e lições que foram mais que valiosas para confecção deste trabalho. Ao professor e maior líder servidor que já conheci nesta vida, Prof. Dr. Guilherme Augusto Barucke Marcondes, que me inspira desde os primeiros períodos da faculdade com sua postura e profissionalismo inigualável, obrigado por acreditar no meu potencial e por me apoiar incondicionalmente em minhas iniciativas. E claro, meu maior respeito e admiração dedico ao grande responsável por tudo que aconteceu, Prof. Dr. Enzo Seraphim, que de forma maestral me auxiliou e me conduziu no decorrer deste desafio. Assim como um operador de junção “combina tuplas de duas relações diferentes para se criar uma nova relação resultado”, aqui, combinamos nossas forças e conhecimentos para criarmos juntos este trabalho.

Sumário

1 Introdução **1**

1.1 Motivação 2

1.2 Objetivos 3

1.3 Organização do Trabalho 3

2 Fundamentação Teórica **4**

2.1 Considerações Iniciais 4

2.2 Métodos de Acessos Métricos 4

2.2.1 M-Tree 6

2.3 Métodos de Acessos Espaciais 8

2.3.1 R-Tree 9

2.4 O Framework Object-Injection 10

2.4.1 O Módulo Metaclasses 11

2.4.2 O Módulo de Armazenamento 12

2.4.3 O Módulo de Blocos 13

2.4.4 O Módulo de Dispositivos 16

2.5 Junção 17

2.5.1 Junção por Abrangência (ε-*Join*) 18

2.5.2 Junção por Vizinhos mais Próximos (*kNN-Join*) 19

2.5.3 Junção por Proximidade (*kD-Join*) 20

2.5.4 Junção ao Redor (*A-Join*) 21

2.6 Junção Canalizada (*Channeled-Join*) 23

2.7 Trabalhos Relacionados 24

2.8 Considerações Finais 27

3 Aprimoramentos da Junção Canalizada aplicada em dados Métricos e Espaciais **29**

3.1 Considerações Iniciais 29

3.2 Definição da Junção Canalizada ($\bowtie$) 29
3.3 Algoritmos para a Junção Canalizada 32

4 Experimentos e Resultados **42**
4.1 Experimento 1 (UNIFORME) 44
4.2 Experimento 2 (UK Postcodes) 46
4.3 Experimento 3 (ALOI) 48

5 Conclusão **52**

Lista de Figuras

2.1 Estrutura dos nós da *M-Tree*. Adaptada de (TRAINA et al., 2002) 7

2.2 Objeto de roteamento na estrutura da M-Tree. Adaptada de (SKOPAL et al., 2003) . 8

2.3 Hierarquia de regiões métricas na *M-Tree*. Adaptada de (SKOPAL et al., 2003) . . . 8

2.4 Representação da Árvore R. (MANOLOPOULOS et al., 2005) 9

2.5 Organização e interação dos módulos do *framework Object-Injection*. (CARVALHO, 2013) 11

2.6 Classes do pacote Metaclasses do *framework Object-Injection*. Adaptada de (FERRO, 2012) 11

2.7 Classes do pacote Armazenamento do *framework Object-Injection* (FERRO, 2012) . 13

2.8 Estrutura geral de um bloco no *framework Object-Injection* (CARVALHO, 2013) . . 14

2.9 Classes do pacote Blocos do *framework Object-Injection* (FERRO, 2012) 14

2.10 Organização dos blocos da estrutura de dados *M-Tree* no *framework Object-Injection* (CARVALHO, 2013) 15

2.11 Classes do pacote Dispositivos do *framework Object-Injection* (FERRO, 2012) . . . 16

2.12 Junção por Abrangência e seu processo de comutatividade. 19

2.13 Junção por Vizinhos mais Próximos com k = 2 e a não comutatividade desta junção. . 20

2.14 Junção por Proximidade com k = 2 e seu processo de comutatividade. 21

2.15 Junção ao Redor com k = 2, $r = 1$, e seu processo de comutatividade. 22

2.16 (a) Dimensões do Canal, (b) distância de um ponto a para a rota. Adaptada de (DUARTE, 2012) 23

3.1 Regiões da Junção Canalizada. Adaptado de (DUARTE, 2012) 30

3.2 Organização dos blocos índices e folhas na *M-Tree* e interação entre rotas e pontos de interesse na Junção Canalizada. 33

3.3 Exemplo de operação da poda por desigualdade triangular. 38

3.4 Interação entre rotas e pontos do domínio da Junção Canalizada na *R-Tree*. 40

4.1 Experimento 1 (Uniforme) - Média dos tempos de execução, número de verificações nas estruturas de dados e quantidade de acessos a disco de cada um dos algoritmos. . 45
4.2 Experimento 2 (UK *Postcodes*) - Média dos tempos de execução, número de verificações nas estruturas de dados e quantidade de acessos a disco de cada um dos algoritmos. 48
4.3 Experimento 3 (ALOI) - Média dos tempos de execução, número de verificações nas estruturas de dados e quantidade de acessos a disco de cada um dos algoritmos. . . . 50

Lista de Tabelas

4.1 Informações relevantes sobre a construção das estruturas utilizadas pelos algoritmos da junção canalizada. 43

Resumo

Dentre as diversas operações de bancos de dados relacionais disponíveis para especificar consultas de dados, a junção é uma operação que tem sido largamente discutida e estudada na literatura, pois se apresenta como uma das operações que mais consomem tempo de processamento e que mais fazem uso dos conjuntos de dados envolvidos. Desde o surgimento dos bancos de dados relacionais, a junção tem sido extensamente utilizada para resolver diversos tipos de problemas que envolvam conjuntos de dados, sejam eles com relação de ordem total, métricos ou espaciais. No meio dos diversos operadores de junção que foram propostos para lidarem principalmente com dados métricos ou espaciais, uma nova operação de junção denominada Junção Canalizada, que pode ser largamente utilizada na busca por objetos de interesse ao longo de rotas, ainda não foi completamente formalizada e foi pouco explorada pela literatura. O objetivo deste trabalho é formalizar o operador de junção denominado Junção Canalizada por meio do cálculo relacional, e principalmente, explorá-lo mais em detalhes por meio da apresentação de cinco novos algoritmos que visam realizar melhorias sobre ele.

Palavras-chave: Junção Espacial, Junção Métrica, Consulta Contínua.

Abstract

Among the several relational databases operations available to specify data queries, the join operator has been widely discussed and studied in the literature, once it is one of the operations that most consume processing time and that more make use of the data sets involved. Since the emergence of relational databases, the join operator has been widely used to solve many types of problems involving data sets, independent if they are of a total order, metric or spatial. Among the various join operators that have been proposed to deal primarily with metric or spatial data, a new join operator called Channeled Join, which can be widely used to search for objects of interest along routes, has not yet been completely formalized and was little explored in the literature. The objective of this work is to formalize the join operator called Channeled Join through relational calculus, and mainly, explore it more in detail through the presentation of five new algorithms that aim to perform improvements on this operator.

Keywords: Spatial Join, Metric Join, Continuous Query.

Capítulo
1
Introdução

Desde o começo da humanidade, o homem busca armazenar e consultar dados. Inicialmente, as pinturas rupestres se apresentaram como principal meio de armazenamento. Por meio delas, era possível compartilhar com outras pessoas e gerações diversos detalhes de como os homens viviam e resolviam problemas do seu cotidiano. Em seguida, diversas outras formas de armazenamento de dados foram surgindo a fim de facilitar ainda mais a transferência de conhecimento, como tabletes de argila, pergaminhos, o próprio papel, e por fim, o armazenamento eletromagnético.

Dentre os diversos tipos de armazenamento de dados citados, o armazenamento eletromagnético se destaca como o mais rápido e duradouro, uma vez que são interpretados por dispositivos eletrônicos velozes e que possuem um tempo de vida maior quando armazenados de forma correta. O processamento de dispositivos eletromagnéticos pode ser feito de forma analógica ou digital, porém, atualmente a grande maioria dos dispositivos são do tipo digital, pois além de serem mais rápidos, muitas vezes também são mais baratos que os dispositivos analógicos.

No meio de armazenamento digital, a primeira forma de armazenamento que se popularizou foi o arquivo. Apesar deste meio ser bastante simples, ele possui grandes restrições, principalmente nos quesitos segurança e acesso simultâneo de dados. Por conta destes e outros fatores, tais como o advento da tecnologia e o aumento da importância do gerenciamento dos sistemas de informação, em meados de 1970 Ted Codd da IBM apresentou a ideia de bancos de dados relacionais.

Atualmente, os bancos de dados relacionais são uma das formas de armazenamento de dados mais populares do mercado. De forma geral, eles organizam suas informações em tabelas que se relacionam, buscando organizá-las da forma mais prática possível. Sem contar que também oferecem recursos formais que facilitam a construção de consultas que podem ser realizadas sobre os dados contidos nas tabelas, como é o caso da álgebra relacional.

A álgebra relacional constitui o conjunto básico de operações que podem ser utilizadas em bancos de dados relacionais para especificar a solicitação básica de recuperação de dados. Suas operações

podem ser divididas em dois grupos: um grupo inclui um conjunto de operações da teoria de conjuntos da matemática como união, interseção, diferença e produto cartesiano e o outro grupo consiste em operações desenvolvidas especialmente para bancos de dados relacionais como seleção, projeção e junção (ESMASRI; NAVATHE, 2005). Dentre as diversas operações citadas, o objeto de estudo deste trabalho é a junção.

1.1 Motivação

A junção é uma importante operação de banco de dados relacionais, pois combina valores de dois ou mais conjuntos de dados se baseando em alguma informação em comum entre eles. Esta operação tem sido largamente discutida e estudada na literatura por conta de ser uma das operações que mais consomem tempo de processamento e que mais fazem uso dos conjuntos de dados envolvidos. Além disso, a junção pode ser implementada de diferentes maneiras, e é sabido que certas técnicas utilizadas para implementá-las são mais eficientes do que outras em determinados casos (MISHRA; EICH, 1992).

Desde o surgimento dos bancos de dados relacionais, em meados da década de 70, a junção tem sido extensamente utilizada para resolver diversos tipos de problemas que envolvam conjuntos de dados. Porém, por muito tempo essa operação ficou restrita apenas a conjuntos de dados com relação de ordem total entre seus elementos. De forma geral, estes conjuntos de dados são capazes de serem comparados utilizando-se dos operadores $<, >, \leq, \geq, = e \neq$ e contemplam apenas domínios de dados númericos e pequenas sequências de caracteres.

Com o advento e popularização de dispositivos capazes de produzirem novos tipos de dados, novas formas de buscas também tiverem de ser criadas e exploradas a fim de adaptarem estes dados que não seguem uma relação de ordem total aos bancos de dados relacionais. Estes dados foram chamados de dados complexos, e uma forma de compará-los é utilizando-se de uma função de dissimilaridade, que nada mais é que uma função que retorna o quanto um elemento do conjunto é diferente do outro. A busca por semelhança ou por proximidade se tornou uma tarefa computacional fundamental em uma variedade de áreas, incluindo busca de dados multimídia (como fotos e vídeos), mineração de dados, reconhecimento de padrões, aprendizado de máquina, visão computacional, compressão de dados, entre outras (ZEZULA; AMATO, 2006).

Dentre os diversos operadores de junção que foram propostos para lidarem com dados métricos, uma nova operação de junção denominada Junção Canalizada (DUARTE, 2012), que pode ser largamente utilizada na busca por objetos de interesse ao longo de rotas, ainda não foi completamente formalizada e foi pouco explorada pela literatura. Por conta disso, se torna um objeto alvo e potencial de novos estudos.

1.2 Objetivos

O objetivo deste trabalho é formalizar o operador de junção denominado Junção Canalizada por meio do cálculo relacional e principalmente explorá-lo mais em detalhes por meio da implementação de novos algoritmos. A proposta é apresentar melhorias à este operador de junção que usa de conjuntos de dados métricos ou espaciais que operam sobre a geometria euclidiana. Por se tratar de um operador de junção dependente de uma estrutura de indexação em disco dos dados envolvidos, as estruturas de dados *R-Tree* e principalmente *M-Tree* serão utilizadas para darem suporte na implementação dos diferentes algoritmos propostos.

1.3 Organização do Trabalho

No segundo capítulo inicialmente serão abordados os fundamentos e ferramentas que serviram de base para construção deste trabalho. Este capítulo começa explorando fundamentos importantes relacionados aos dados métricos e espacias, tão bem como detalhes de operação de uma estrutura de dados métrica e uma estrutura de dados espacial bastante populares nestes meios, neste caso, as árvores de indexação *M-Tree* e *R-Tree*, respectivamente. Em seguida, são discutidos os princípios de funcionamento e operação do *framework Object-Injection*, responsável por abstrair e facilitar a utilização das estruturas de dados citadas nos algoritmos implementados. Logo adiante, é apresentado o conceito geral de junção e detalhes de funcionamento dos principais operadores de junção por similaridade utilizados em espaços métricos. Dando sequência, o conceito geral de junção canalizada é explicado, oferecendo assim uma visão geral de funcionamento deste operador, que é objeto de estudo deste trabalho. Finalmente, neste capítulo, trabalhos renomados da literatura e que usam de recursos parecidos com os utilizados na junção canalizada são discutidos e revisados a fim de identificar possíveis semelhanças entre as soluções.

No terceiro capítulo é apresentada a formalização da junção canalizada utilizando de operadores do cálculo relacional e detalhes de implementação de cada um dos algoritmos propostos são discutidos, buscando mostrar a evolução e aplicação de cada um deles na busca de pontos de interesse ao longo de uma rota.

No quarto capítulo, experimentos mostram como foi o comportamento de cada um dos algoritmos propostos quando sujeitos a conjuntos de dados sintéticos, semi-sintéticos e reais.

Por fim, no último capítulo, são apresentadas as conclusões e contribuições deste trabalho, assim como também possíveis propostas de trabalhos futuros.

Capítulo
2
Fundamentação Teórica

2.1 Considerações Iniciais

Neste capítulo são apresentados os principais conceitos que serviram de base para o desenvolvimento deste trabalho. Primeiramente, são discutidos conceitos de dados métricos e espaciais. Estes conjuntos de dados são extremamente importantes nos dias de hoje, pois permitem a indexação e manipulação de elementos complexos como coordenadas geográficas, sequência de proteínas, textos longos, imagens, entre outros. Uma estrutura de dados importante para indexação de dados métricos como a *M-Tree* (principal estrutura explorada neste trabalho) e uma estrutura de dados bastante popular para indexação de dados espaciais denominada *R-Tree*, também são exploradas.

Na sequência, é abordado o princípio de funcionamento do *framework Object-Injection*, que se apresenta como uma solução orientada a objetos que facilita a construção e manipulação de diversas estruturas de dados de forma mais prática, fazendo com que o desenvolvedor foque na lógica de sua aplicação e não nos detalhes de implementação destas estruturas.

Em seguida, é discutido o conceito de junções, operações de buscas fundamentais utilizadas em bancos de dados, conhecidas por serem computacionalmente caras mas que são capazes de combinar vários conjuntos de dados a fim de obter um novo conjunto resposta.

A seção seguinte aborda os conceitos fundamentais de operação da Junção Canalizada (principal objeto de estudo deste trabalho), e por fim, este capítulo se encerra apresentando e discutindo soluções que utilizam de recursos parecidos com a da junção canalizada para realizar diversos tipos de operações sobre dados métricos e espaciais.

2.2 Métodos de Acessos Métricos

O armazenamento e a busca de dados são funcionalidades que nortearam o desenvolvimento dos sistemas gerenciadores de banco de dados (SGBD). Porém, grande parte destes gerenciadores foram inicialmente desenvolvidos para manipularem domínios de dados numéricos e sequência de caracte-

res, ou seja, dados que possuem uma relação de ordem total entre seus elementos. Estes dados têm como característica utilizarem dos operadores relacionais =,$\neq$, <, >, $\leq$, $\geq$ para realizarem comparações entre seus elementos e devem obedecer às seguintes propriedades:

- **Transitividade:** $\forall a, b, c \in S, a \leq b \wedge b \leq c \Rightarrow a \leq c$;
- **Anti-simetria:** $\forall a, b \in S, a \leq b \wedge b \leq a \Rightarrow a = b$;
- **Totalidade:** $\forall a, b \in S \Rightarrow a \leq b \vee b \leq a$;

Apesar dos dados com relação de ordem total possuirem inúmeras aplicações e serem de extrema relevância, eles apresentam restrições e não podem ser usados para representarem ou manipularem novas coleções de dados presentes nesta era digital, como por exemplo, imagens, áudios, vídeos, textos longos, sequências de proteínas, entre outros. A este novos tipos de dados, ou seja, dados que não possuem uma relação de ordem total entre seus elementos, dá-se o nome de dados complexos. Estes dados, bem diferente dos dados com relação de ordem total, possuem uma característica importante que é a possibilidade de verificar o quanto um elemento é parecido ou diferente do outro. A dissimilaridade, por exemplo, é um valor que mede o quanto dois elementos de um mesmo domínio são diferentes um do outro. A estes conjuntos de dados que utilizam da dissimilaridade para comparar seus elementos, dá-se o nome de conjunto de dados métricos. Formalmente, dois objetos x, y pertencentes a um conjunto de dados métricos podem se comparar fazendo uso de uma função de distância (ou dissimilaridade) $d()$ que satisfaça as seguintes propriedades (TRAINA et al., 2002):

- **Simetria:** $d(x, y) = d(y, x)$;
- **Não Negatividade:** $0 < d(x, y) < \infty, x \neq y \; e \; d(x, x) = 0$;
- **Desigualdade Triangular:** $d(x, y) \leq d(x, z) + d(z, y)$;

Sobre estes dados, consultas por similaridade podem ser aplicadas para avaliarem o grau de dissimilaridade entre seus objetos e retornarem, se baseando em determinados parâmetros, aqueles objetos mais parecidos com um objeto de consulta. Os dois tipos de consultas por similaridade mais comuns são a consulta por abrangência e a consulta por k-vizinhos mais próximos. A consulta por abrangência pesquisa por objetos que estão dentro de uma determinada distância em relação a um objeto de consulta, como por exemplo: "Encontre as estrelas que estão dentro de um raio de 10 anos luz do sol". Já a consulta por k-vizinhos mais próximos busca pelos n objetos mais próximos do objeto de consulta, por exemplo: "Selecione as 5 estrelas mais próximas do sol"(TRAINA et al., 2002).

Vale ressaltar que em alguns casos, antes da realização de uma consulta por similaridade, inicialmente os dados a serem utilizados na pesquisa precisam estar indexados em uma estrutura de dados apropriada. Estruturas de dados capazes de indexarem dados métricos são chamadas de MAMs (*Metric Access Methods*).

Uma MAM organiza uma grande quantidade de dados métricos permitindo a realização de inserções, atualizações, exclusões e buscas sobre ela. A eficiência de uma MAM pode ser determinada se baseando em diversos fatores, como por exemplo, acessos a discos realizados para processamentos de inserções e buscas, custo computacional dos cálculos de distância, quantidade de armazenamento, entre outros. Árvores métricas são exemplos clássicos de MAMs utilizadas para indexação de dados métricos.

A estrutura básica das árvores métricas visa particionar o espaço de dados em regiões que utilizam de nós representativos ou centrais em que outros objetos podem estar associados. Cada partição tem um raio de cobertura e apenas objetos dentro desse raio de cobertura são associados a um nó representativo (TRAINA et al., 2002). Como exemplo de árvores métricas pode-se citar a *M-Tree* (CIACCIA; PATELLA; ZEZULA, 1997), principal MAM que será utilizada para realização das implementações deste trabalho.

2.2.1 M-Tree

A *M-Tree*, apresentada por Ciaccia, Patella e Zezula (1997), é uma estrutura de dados dinâmica para indexação de objetos pertencentes a um conjunto de dados métricos. A estrutura dessa árvore foi primeiramente desenvolvida para bancos de dados multimídia a fim de suportarem nativamente as buscas por similaridades.

Um espaço métrico é representado por $\mathcal{M} = (D, d)$, em que D é um domínio de objetos características e d é uma função que mede a distância entre dois objetos características. Um objeto característica $O_i \in D$ é uma sequência de características extraídas de um banco de dados do objeto original. A função d deve ser métrica, isto é, d deve satisfazer as propriedadades explicadas anteriormente que são simetria, não negatividade e desigualdade triangular.

A *M-Tree* é baseada em uma organização hierárquica de objetos características de acordo com uma dada métrica d. Do mesmo jeito que outras árvores dinâmicas e persistentes trabalham, a estrutura da *M-Tree* é uma hierarquia balanceada de nós. Geralmente, os nós possuem uma capacidade fixa e um limiar de utilização. Dentro da hierarquia da *M-Tree*, os objetos são agrupados em regiões métricas.

Os nós folhas são os nós que armazenam os objetos em si, enquanto entradas representando as regiões métricas são armazenadas nos nós mais internos, denominados índices (SKOPAL et al., 2003). Um objeto do tipo folha O_i possui o seguinte formato:

$$folha(O_i) = [O_i, oid(O_i), d(O_i, O_{rep})],$$

em que $O_i \in D$ é o objeto característica, $oid(O_i)$ é um identificador do objeto no banco de dados (armazenado externamente), e $d(O_i, O_{rep})$ é uma distância pré-calculada entre O_i e seu objeto representativo (ou objeto de roteamento pai). Já um objeto do tipo índice possui o seguinte formato:

$$indice(O_j) = [O_j, ptr(T(O_j)), r(O_j), d(O_j, O_{rep})],$$

em que $O_j \in D$ é o objeto característica, $ptr(T(O_j))$ é um ponteiro para uma subárvore de cobertura, $r(O_j)$ é um raio de cobertura, e $d(O_j, O_{rep})$ é uma distância pré-calculada entre O_j e seu objeto representativo (este valor é zero para os objetos de roteamento armazenados na raiz).

A Figura 2.1 ilustra como é formado os nós da *M-Tree*.

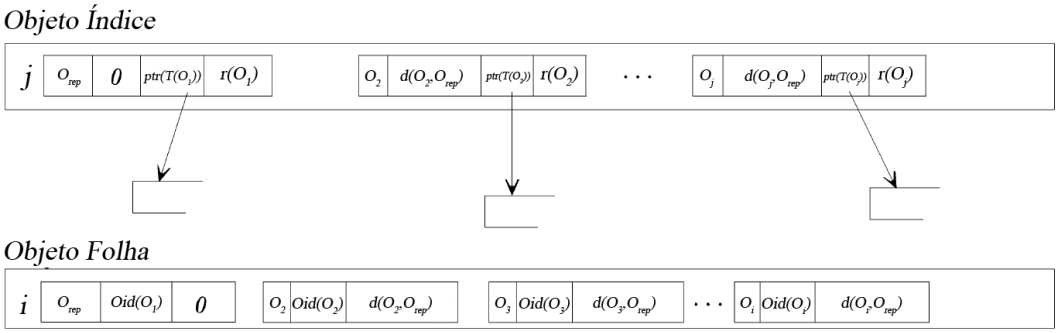


Figura 2.1: Estrutura dos nós da *M-Tree*. Adaptada de (TRAINA et al., 2002)

A entrada de um objeto de roteamento determina uma região métrica no espaço $\mathcal{M}$ em que o objeto O_j é o centro dessa região e $r(O_j)$ é um raio delimitador. O valor pré-calculado $d(O_j, O_{rep})$ é redundante e serve para otimizar os algoritmos baseados na *M-Tree*.

Na Figura 2.2, uma região métrica e sua entrada apropriada $indice(O_j)$ na *M-Tree* é apresentada. Para a hierarquia de regiões métricas na *M-Tree*, apenas uma propriedade deve ser satisfeita: "Todos os objetos do tipo folha são armazenados no último nível da sub-árvore de cobertura de $indice(O_j)$ e deve estar espacialmente localizado dentro da região definida por $indice(O_j)$." Formalmente, ter um $indice(O_j)$ significa $\forall O \in T(O_j), d(O, O_j) \leq r(O_j)$. Percebe-se que essa propriedade é muito fraca uma vez que podem ser construídas várias *M-Trees* do mesmo conteúdo do objeto mas de estruturas diferentes. A consequência disso é que várias regiões em um mesmo nível da *M-Tree* podem se sobrepor.

Um exemplo na Figura 2.3 ilustra vários objetos particionados em regiões métricas em uma *M-*

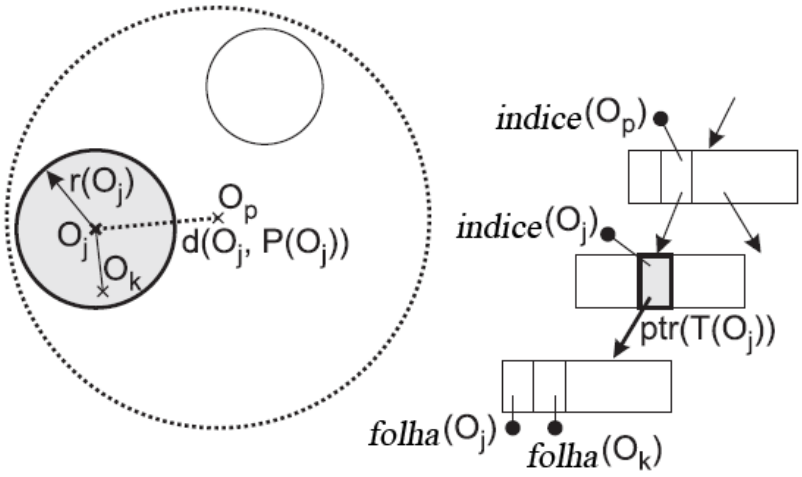


Figura 2.2: Objeto de roteamento na estrutura da M-Tree. Adaptada de (SKOPAL et al., 2003)

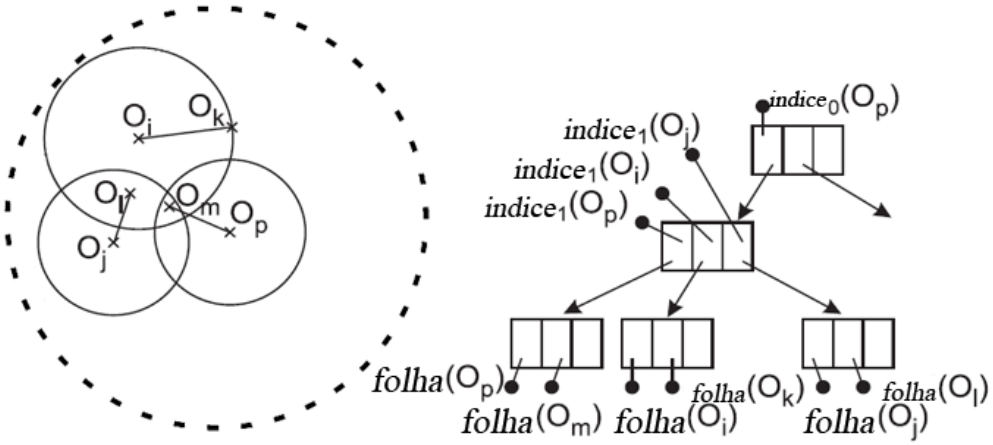


Figura 2.3: Hierarquia de regiões métricas na *M-Tree*. Adaptada de (SKOPAL et al., 2003)

Tree. Como pode ser visto, as regiões definidas por $indice_1(O_p), indice_1(O_i)$ e $indice_1(O_j)$ se sobrepõem. Além disso, o objeto O_l está localizado dentro das regiões de $indice(O_i)$ e $indice(O_j)$ mas ele é armazenado apenas na subárvore de $indice_1(Oj)$. Da mesma forma, o objeto O_m está localizado em três regiões, mas esta armazenado apenas na subárvore de $indice_1(O_p)$. A sobreposição de regiões na *M-Tree* é um problema sério que pode chegar a degradar o desempenho de uma busca por similaridade sobre esta estrutura.

2.3 Métodos de Acessos Espaciais

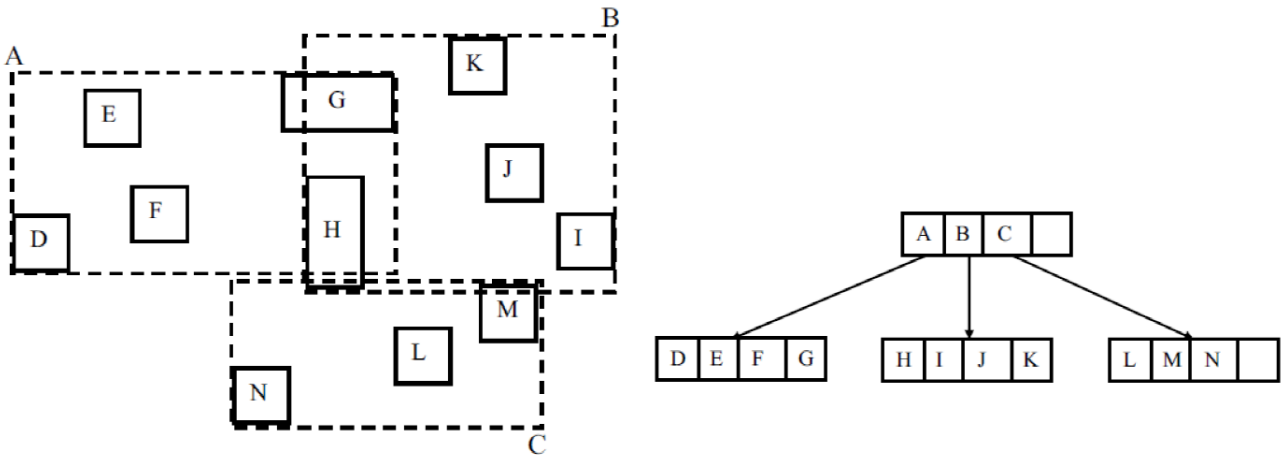
Dados espaciais consistem de objetos como pontos, retas, regiões, retângulos, superfícies ou até mesmo dados de várias dimensões, que podem ou não incluir o tempo (KIM, 1995). Como exemplo de dados espaciais, pode-se citar diferentes tipos de localidades, como os limites de uma cidade, a

extensão de uma estrada ou rio, o tamanho de uma plantação, entre outros.

Diferente das formas de organização de dados utilizadas por grande parte dos Sistemas Gerenciadores de Banco de Dados tradicionais, a organização dos dados espaciais deve ser baseada em suas chaves espaciais, ou seja, a indexação é realizada de acordo com o espaço ocupado pelo dado. Esta técnica, denominada Método de Acesso Espacial ou Multidimensional, foi desenvolvida para indexar um conjunto de dados vetoriais, a fim de processar de forma eficiente consultas pontuais, por abrangência, entre outras. A seguir, será discutido os detalhes de operação da *R-Tree*, que se apresenta como uma estrutura de dados multidimensional largamente utilizada para indexação de objetos espaciais e que também é objeto de estudo deste trabalho.

2.3.1 R-Tree

Semelhante à *B-Tree*, a *R-Tree* é uma estrutura de dados hierárquica, dinâmica e balanceada em altura, porém, utilizada para indexar dados espaciais de maneira eficiente, permitindo que os mesmos sejam recuperados rapidamente de acordo com suas localizações no espaço. Como discutido, estes dados assumem o papel de objetos geométricos, mas que são envolvidos pelos chamados retângulos delimitadores mínimos *(MBR - Minimum bounding rectangle)*. Cada MBR está associado a um único objeto que é incluído em um único nó da árvore, porém, um MBR pode se sobrepor a outro, como ilustra a Figura 2.4(a). Esta Figura também ilustra um conjunto de MBRs bidimensionais de objetos de dados geométricos. Os três MBRs maiores, representados pelas letras A,B e C, são nós internos da árvore que organizam os nós menores ou nós folhas, representados pelas letras D,E,F,G,H,I,J,K,L,M e N. Já a Figura 2.4(b) ilustra a *R-Tree* correspondente ao conjunto de MBRs da Figura 2.4(a), porém, em formato de árvore.



(a) Conjunto de MBRs (b) Árvore R correspondente

Figura 2.4: Representação da Árvore R. (MANOLOPOULOS et al., 2005)

Segundo Manolopoulos et al. (2005), as principais características da *R-Tree* são:

- Todo nó da árvore, seja folha ou índice, tem capacidade para M entradas. Caso o nó não seja a raiz, o número mínimo de entradas permitido é $m = \lfloor M/2 \rfloor$. O limite inferior m previne a degeneração da árvore e garante uma utilização eficiente do espaço de armazenamento. O limite superior M baseia-se no fato de cada nó corresponder exatamente a uma página de disco;

- A raiz contém, no mínimo, duas entradas, exceto se for uma folha, podendo conter uma única entrada ou nenhuma, caso a árvore esteja vazia;

- Cada entrada em um nó interno tem a forma (mbr, ptr), em que ptr é o apontador para a sub-árvore correspondente e mbr é o MBR que cobre toda esta sub-árvore;

- Todas as folhas estão no mesmo nível e cada entrada tem a forma (mbr, oid), em que mbr é o MBR que armazena o objeto e oid é o identificador do objeto;

Segundo Guttman (1984), as operações de busca mais utilizadas em uma *R-Tree* são as consultas por abrangência (*range query*), que encontram todos os objetos que interceptam uma região de busca, e pontuais (*point query*), que faz o mesmo, porém, com um ponto no espaço. O algoritmo de busca percorre a árvore, a partir da raiz, de forma similar a uma árvore B, porém aqui, em cada nó índice, todas as entradas devem ser testadas para verificar se existe alguma região de interseção com a região de busca. Devido às interseções entre os MBRs, pode ser necessário que mais de uma subárvore, em cada nível, necessite ser visitada até atingir as folhas. Por sua vez, as entradas das folhas devem ser testadas, verificando a interseção com a região ou ponto buscado.

2.4 O Framework Object-Injection

O *Object-Injection* (CARVALHO et al., 2013) é um *framework* orientado a objetos de indexação e persistência de objetos que permite ao desenvolvedor injetar objetos em diversas estruturas de dados que podem estar localizadas em memória principal ou secundária. Uma vez geradas as classes de empacotamento a partir das classes da aplicação, o *framework* possibilita a persistência de objetos em diferentes estruturas de dados de forma nativa e oferece ao programador mais liberdade para se concentrar na lógica de negócio do programa, sem se preocupar com os detalhes de como os objetos serão armazenados.

Este *framework* é particionado em quatro módulos (ou pacotes) de abstração, como pode ser visto na Figura 2.5. Cada módulo é responsável por uma tarefa específica no processo de persistância de dados. Estes módulos são os seguintes:

- **Metaclasses:** define o vínculo entre as classes da aplicação do usuário que desejam se tornarem persitentes e o *framework*;

- **Blocos:** define a maneira pela qual as entidades persistentes e chaves indexadas são armazenadas em blocos ou páginas no disco;
- **Armazenamento:** define estruturas de índices primários para as entidades persistentes e estruturas de índices secundários para as chaves indexadas. É também o módulo responsável por gerenciar e organizar os dados do módulo Metaclasses, que são mantidos no módulo Blocos;
- **Dispositivos:** define os recursos computacionais responsáveis pelo armazenamento físico das estruturas de índices. Além disso, este módulo fabrica blocos requisitados pelo módulo de Armazenamento.

Nos subtópicos a seguir cada um destes módulos e suas responsabilidades serão explanados em mais de detalhes.

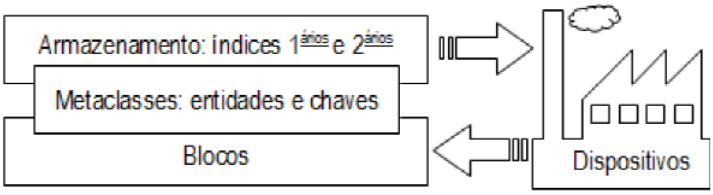


Figura 2.5: Organização e interação dos módulos do *framework Object-Injection*. (CARVALHO, 2013)

2.4.1 O Módulo Metaclasses

Todas as classes da aplicação que desejam utilizar da persistência oferecida por este *framework* devem implementar uma interface comum, neste caso, a interface *Entity*. A Figura 2.6 ilustra a classe *City* que foi definida pelo usuário em uma aplicação específica e pela qual ele deseja tornar persistente.

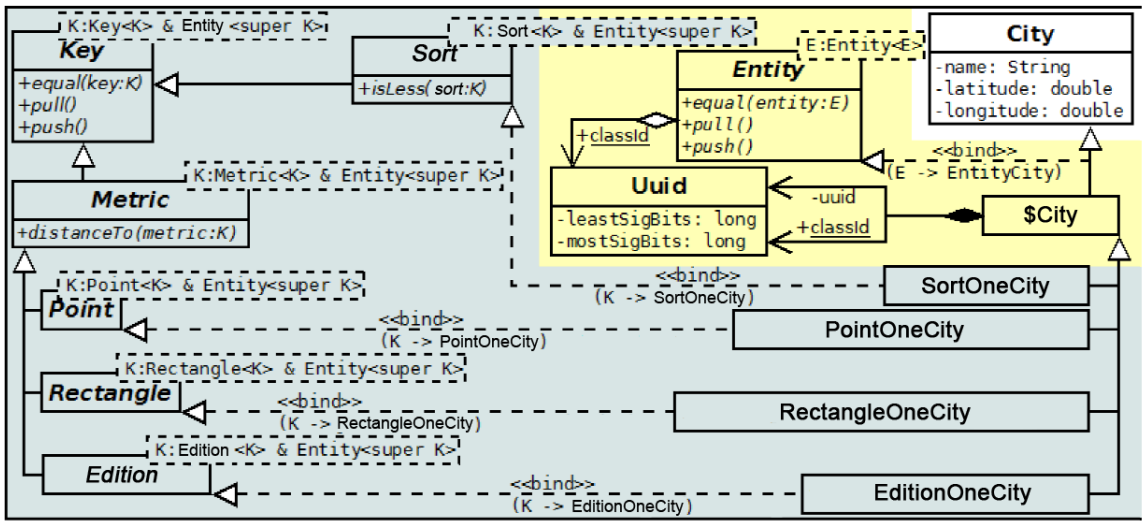


Figura 2.6: Classes do pacote Metaclasses do *framework Object-Injection*. Adaptada de (FERRO, 2012)

Como pode ser observado, por meio do uso de um conjunto de interfaces e heranças, o *framework* é capaz de gerar outras classes automaticamente ao usuário sem que haja necessidade de alterações na sua classe original (neste caso, a classe *City*), oferecendo assim, um alto grau de desacoplamento entre as classes. Por exemplo, a classe *$City* especializa a classe concreta *City* e implementa os métodos da interface *Entity*. Assim, *$City* possui as mesmas características da classe *City* e também implementa os métodos requeridos pela interface *Entity*. É importante ressaltar que a interface *Entity* é uma classe parametrizada e isto é feito para garantir que somente objetos de uma mesma classe possam ser comparados uns com os outros.

Somente entidades persistentes podem ter domínios de chave associados a seus atributos. Para definir um domínio de chave, uma entidade deve implementar uma das interfaces providas pela classe *Key*. Os métodos da classe *Key* são idênticos aos da classe *Entity*, porém, enquanto a *Entity* compara e serializa todo o conjunto de atributos da classe, uma *Key* realiza o mesmo procedimento, mas apenas com o atributo de indexação (CARVALHO, 2013). A classe *Key* se especializa em *Sort* e *Metric*. Chaves derivadas de *Sort* indexam um conjunto de objetos que satisfazem a relação de ordem total (ou seja, obedecem as propriedades de transitividade, anti-simetria e totalidade). De forma similar, as chaves derivadas de *Metric* indexam um conjunto de objetos que não possuem relação de ordem total mas que podem ser relacionados se baseando em uma função de distância métrica (ou seja, obedecendo as propriedades de simetria, não-negatividade e desigualdade triangular). Na Figura 2.6, a classe *SortOneCity* é uma chave (*Key*) para um objeto *City*, em que o atributo de indexação é o nome da cidade. Nesta classe, o método *isLess()* recebe como parâmetro uma instância de outro objeto do tipo *SortOneCity* e compara se seu atributo *name* vem lexicograficamente antes do atributo *name* do argumento (comparação por meio de uma relação de ordem total). Já as classes *PointOneCity*, *RectangleOneCity* e *EditionOneCity* são chaves para um objeto *City*, em que o atributo de indexação são pontos, retângulos e distância entre *Strings*, respectivamente. Nestas classes, o método *distanceTo()* é utilizado para computar o quão distante uma cidade está da outra, de acordo com uma distância Euclidiana (ou seja, obedecendo as propriedades do espaço métrico).

2.4.2 O Módulo de Armazenamento

Este módulo especifica a maneira pela qual as estruturas de índices primários e secundários são implementados no *framework*. Neste módulo, as estruturas de dados manipulam coleções de blocos, a fim de organizar e armazenar os objetos definidos no módulo de Metaclasses. A Figura 2.7 ilustra as classes deste módulo e suas responsabilidades.

Antes que o usuário possa fazer uso das classes deste módulo, ele precisa primeiramente criar um *Workspace*, ou seja, uma área virtual que as estruturas de dados podem usar para acessar e comparti-

lhar dados. Esta área é criada por meio de uma classe que se encontra dentro do módulo Dispositivos e será explicada mais em detalhes na seção 2.4.4. Uma vez criada uma *Workspace*, a classe *Structure* no topo da hierarquia já pode ser criada para permitir a criação, recuperação e atualização de blocos por meio do *framework*. A classe *Structure* é especializada em *EntityStructure* e *KeyStructure*. A classe *EntityStructure* é um índice primário responsável por manipular objetos inteiros persistentes e define operações comuns (por exemplo, o *find()*) a todas as estruturas de dados responsáveis pelo armazenamento destes objetos, tais como as classes *Sequential* e *EHash*. Já a classe *KeyStructure* é um índice secundário responsável pelo gerenciamento de chaves indexadas e que define operações comuns para as estruturas de dados responsáveis pelos armazenamentos de índices, tais como as classes *BTree, MTree e RTree* (CARVALHO, 2013).

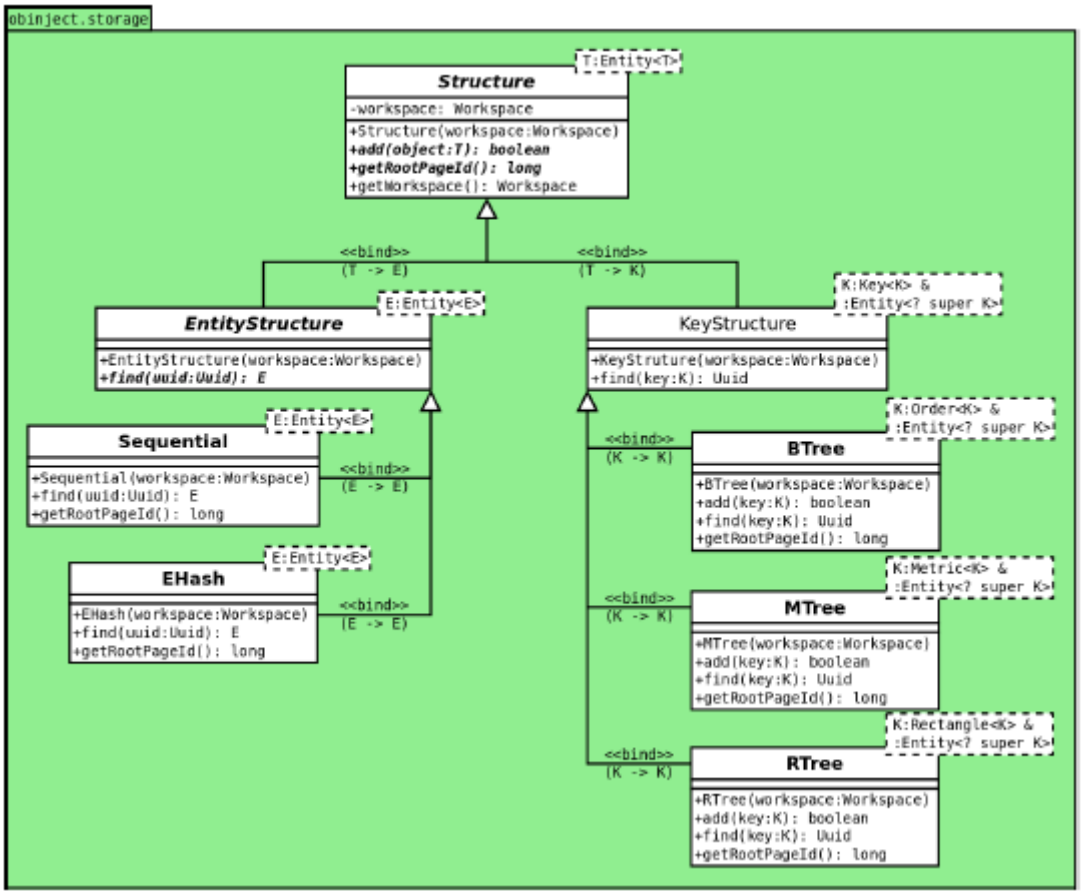


Figura 2.7: Classes do pacote Armazenamento do *framework Object-Injection* (FERRO, 2012)

2.4.3 O Módulo de Blocos

No *Object-Injection*, um bloco (representado por um vetor de *bytes* localizado na memória secundária) é a unidade básica de transferência de dados utilizada para armazenamento das informações nas estruturas de dados. No módulo Blocos é definido as estruturas destes blocos, o formato no qual os dados são armazenados e quais operações são necessárias para armazenamento e recuperação de dados sobre eles. A Figura 2.8 ilustra a estrutura de um bloco utilizado pelo *Object-Injection framework*.

Figura 2.8: Estrutura geral de um bloco no *framework Object-Injection* (CARVALHO, 2013)

De forma geral, todo bloco é dividido em *header* e *free space*. Dentro do *header*, o campo *node type* é representado por um valor inteiro utilizado para determinar de qual estrutura de dados o bloco faz parte, garantindo assim, que uma estrutura de dados não seja capaz de manipular blocos pertencentes a outras estruturas de dados. Já os campos *previous page ID* e *next page ID* são utilizados para permitirem a navegação entre blocos vizinhos em uma lista circular duplamente encadeada de blocos. Por fim, a área de *free space* é usada exatamente para alocar as próprias informações das estruturas de dados.

A fim de que um bloco seja capaz de ser manipulado, ou seja, permita que novas informações sejam armazenadas dentro dele ou que seus dados sejam recuperados, ele deve ser encapsulado em uma estrutura que forneça estas operações. Esta estrutura é a classe *Node*, ilustrada na Figura 2.9. Nesta classe, o atributo *array* representa o bloco no qual os dados serão armazenados e recuperados e diversos métodos de leitura e escrita para tipos primitivos de dados são disponibilizados (por exemplo, *readFloat()* e *writeFloat()* são métodos que permitem respectivamente a leitura e escrita de valores do tipo *float* armazenados em um determinado bloco). Um detalhe importante e que também pode ser visto na Figura 2.9 é que a classe *Node* é abstraída (*AbstractNode*) permitindo que estruturas de dados distintas organizem os *Nodes* de formas diferentes. A classe *AbstractNode* é especializada em *EntityNode, DescriptorNode* e *KeyNode*.

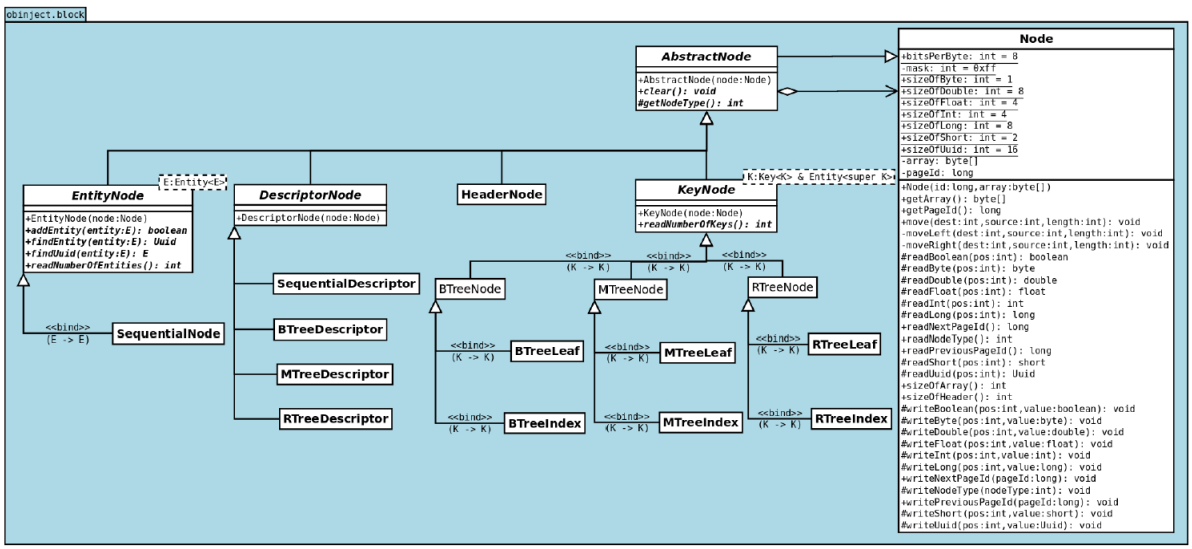


Figura 2.9: Classes do pacote Blocos do *framework Object-Injection* (FERRO, 2012)

As classes *EntityNode* e *KeyNode* armazenam, respectivamente, as entidades persistentes e chaves indexadas e o *DescriptorNode* é um nó especial usado pelas estruturas de dados para armazenarem informações tais como a posição do nó raiz, a lista de blocos livres, valores estatísticos e a altura da árvore (CARVALHO, 2013).

Dentro da área *free space* de um bloco existem outras seções responsáveis pelo armazenamento de informações específicas, que são: a seção *features*, que é a área responsável pelo armazenamento de dados gerais sobre os blocos; *entries*, com informações sobre a localização dos objetos dentro do bloco; *entities/keys*, que são os objetos, e literalmente a *free space*, que é uma área de espaço disponível no bloco e que cresce dinamicamente do modo que novos objetos vão sendo armazenados dentro dele. A Figura 2.10 ilustra, por exemplo, a organização da *free space* da estrutura de dados *M-Tree* oferecida pelo *framework Object-Inject*.

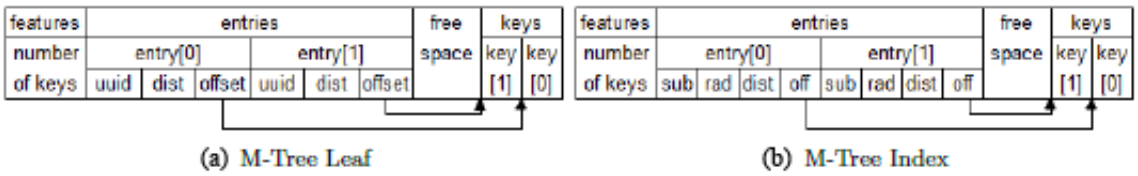


Figura 2.10: Organização dos blocos da estrutura de dados *M-Tree* no *framework Object-Injection* (CARVALHO, 2013)

Os nós da *M-Tree*, principal estrutura de dados envolvida neste estudo, derivam diretamente da classe *KeyNode* e armazenam as chaves indexadas. Estes nós são especializados em nós folha (*MTree-Leaf*), responsáveis pelo armazenamento das chaves inseridas na *MTree* e nós internos (*MTreeIndex*), também chamados de nós índices e responsáveis pelo armazenamento de objetos de roteamento para alcançar estas chaves. As entradas em um nó *MTreeLeaf*, como ilustrado na Figura 2.10 (a), contêm um *uuid* (identificador universal), a distância para o nó representativo (*dist*) e um *offset*. Nos nós do tipo *MTreeIndex* (Figura 2.10 (b)), as entradas contêm um valor *sub* que aponta para uma sub-árvore, o raio de cobertura (*rad*) usado nas operações de inserção e consulta, a distância para a chave representativa (*dist*) e um *offset*. Como pode-se observar, cada objeto armazenado em um nó da *MTree*, seja ele folha ou índice, possui um campo *offset* na seção *entries*. O *offset* referencia o primeiro *byte* do objeto armazenado no fim do bloco e é por meio dele que um objeto de tamanho variável pode ser rapidamente localizado dentro do nó. Entre as seções *entries* e *entities* se encontra a *free space*, usada na inserção de novos objetos. Caso esta área se esgote e não seja mais capaz de armazenar novos objetos, as estruturas de dados devem promover a divisão do nó de acordo com suas próprias políticas de *split*.

2.4.4 O Módulo de Dispositivos

Uma vez que diferentes tipos de dispositivos apresentam características particulares, a maneira mais natural de representá-los é por meio de uma hierarquia de classes abstratas. Este módulo tem como objetivo isolar as estrututas de dados e os objetos persistentes das abstrações de armazenamento. Assim, um objeto não tem conhecimento da existência de dispositivos de armazenamento ou do formato de dados usados na sua persistência (CARVALHO, 2013). Como foi descrito anteriormente, uma *Workspace* nada mais é que uma área virtual que as estruturas de dados podem usar para acessar e compartilhar dados. A classe *AbstractWorkspace* é a interface comum a partir da qual todos os tipos de dispositivos de armazenamento são especializados e que fornece operações primitivas responsáveis por manipularem a *Workspace* destes dispositivos. Como ilustrado na Figura 2.11, esta classe é especializada nas classes *File* e *Memory*. A primeira fornece um tipo de armazenamento de dados em disco e persistente (ou não volátil) e a segunda um tipo de armazenamento especial que é feito na memória RAM (ou seja, um armazenamento volátil).

Por fim, a classe *Session* representa uma sessão de trabalho temporária, que mantém blocos mais recentemente utilizados em uma estrutura de *hash* a fim de acelerar o acesso aos mesmos. Manter na sessão blocos que ainda não foram liberados traz um ganho de desempenho, pois carregá-los a partir de um *File* é uma atividade bem mais lenta. Vale ressaltar que outra tarefa importante desta classe é servir como um objeto intermediário que garante que as estruturas de dados não acessem os métodos de um *AbstractWorkspace* diretamente. Para este propósito, a classe *Workspace* declara o método *openSession()*, que devolve a instância de uma *Session*. Por meio desta instância, uma estrutura de dados torna-se apta a realizar operações nos dispositivos de armazenamento, independente se suas *Workspaces* utilizam de ambientes *File* ou *Memory*.

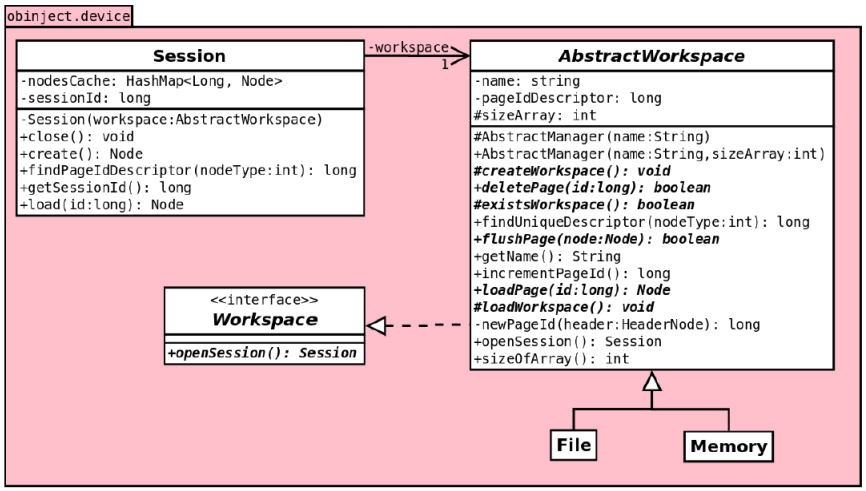


Figura 2.11: Classes do pacote Dispositivos do *framework Object-Injection* (FERRO, 2012)

2.5 Junção

A operação de junção é uma das operações de busca fundamentais utilizadas em bancos de dados relacionais. Esta operação é usada para combinar tuplas de duas relações diferentes baseando em algumas informações em comum entre elas, gerando assim, uma relação resultado. Por serem operações que exigem muitas operações de I/O e que são executadas com bastante frequência nos bancos de dados relacionais, muitas pesquisas têm sido realizadas a fim de melhorar o processamento delas (MISHRA; EICH, 1992). Na forma mais simples, uma junção de R e S é escrita como:

$$R \bowtie_{r\theta s} S,$$

em que $r\theta s$ define a condição da junção. O operador θ define a condição que deve-se manter verdadeira entre os atributos r e s de R e S, respectivamente. Esta junção geral é chamada de *theta-join*. O theta, em um conjunto de dados que obedecem a relação de ordem total, pode ser substituído, por exemplo, por um dos seguintes operadores: =,$\neq$, <, >, $\leq$, $\geq$. Em efeito, a operação de junção é equivalente a um produto cartesiano seguido por uma operação de seleção, em que a operação de seleção está implícita na condição, ou seja:

$$R \bowtie_{r\theta s} S \equiv \sigma_{r\theta s}[R \times S]$$

Como tal, o resultado da junção de duas relações é um subconjunto do produto cartesiano entre elas e o resultado da junção das relações R e S com n e m atributos, respectivamente, é uma relação Q com $(n + m)$ atributos. A relação Q tem uma tupla para cada par de tuplas de R e S que satisfaz a condição de junção. O resultado da relação Q pode então ser definida como:

$$Q = \{t \mid t = r.s \wedge r \in R \wedge s \in S \wedge t(r)\theta t(s)\}$$

A junção por similaridade é uma operação que combina dois conjuntos de dados usando um predicado de junção que conecta tuplas com valores similares. Ela tem sido estudada como um componente chave para resolver diversos tipos de problemas em espaços que não utilizam de elementos com relação de ordem total, como por exemplo, o espaço métrico. A definição genérica para o operador de Junção por Similaridade (JS) é a seguinte:

$$R \bowtie_{\theta_S} S = \{\langle r, s\rangle \mid \theta_S(r, s), r \in R, s \in S\},$$

em que θ_S representa o predicado da junção por similaridade. Este predicado especifica as condições de similaridade que o par $\langle r, s \rangle$ deve satisfazer para ser resultado da junção. Na literatura, existem quatro tipos bem conhecidos de predicados de junção por similaridade (SILVA; AREF; ALI, 2010), são eles:

- **Junção por Abrangência (ou $\varepsilon - Join$):** $\theta_\varepsilon \equiv dist(r, s) \leq \varepsilon$;
- **Junção por Vizinhos mais Próximos (ou *kNN-Join*):** $\theta_{kNN} \equiv$ s em que s é um k-vizinho mais próximo de r;
- **Junção por Proximidade (ou *kD-Join*):** $\theta_{kD} \equiv \langle r, s \rangle$ é um de todos os k-pares mais próximos;
- **Junção ao Redor (ou *A-Join*):** $\theta_{A,MD=2\varepsilon} \equiv$ s que é o vizinho mais próximo de r e $dist(r, s) \leq \varepsilon$.

Os operadores de junção por abrangência, por vizinhos mais próximos e por proximidade são tipos mais comuns de junções por similaridade. Já a junção ao redor é um tipo de junção por similaridade diferente, que combina propriedades da junção por abrangência e por vizinhos mais próximos. Os detalhes de operação de cada um destes predicados serão explicados nos subtópicos a seguir. Além destes quatro tipos de predicados explorados largamente pela literatura, um quinto e novo tipo também será citado e explorado mais em detalhes neste trabalho. Este outro predicado é denominado Junção Canalizada e tem como objetivo realizar a junção de rotas e pontos do domínio que se encontram em um espaço métrico. Esta junção, em específico, será abordada em um tópico separado, exatamente por ser o foco e objeto de estudo deste trabalho.

2.5.1 Junção por Abrangência (ε-*Join*)

Esta junção recebe como parâmetros de entrada dois conjuntos de dados R e S pertencentes a um mesmo domínio e uma distância máxima ε. O predicado de junção busca todos os pares de objetos que estejam à distância ε uns dos outros. Formalmente, esta junção é definida como:

$$R \underset{\varepsilon}{\bowtie} S := \{(r_i, s_j) \in R \times S : dist(r_i, s_j) \leq \varepsilon\}$$

O Algoritmo 1 exemplifica como esta junção pode ser implementada com o uso de *loops* aninhados. Como esta operação é comutativa, ou seja, a ordem das relações não altera o resultado dos pares encontrados desde que se considere o mesmo raio, pode-se usar a relação com a menor cardinalidade no laço externo a fim de melhorar o seu desempenho.

A Figura 2.12 ilustra o funcionamento da junção por abrangência e o seu processo de comutatividade. As estrelas representam os elementos do primeiro conjunto de dados (R), os quadrados representam os elementos do segundo conjunto (S), os círculos representam o raio de cobertura (ε) e as conexões entre as estrelas e os quadrados representam os pontos de junção. Como se pode observar, independente se o conjunto R se encontra no laço mais externo (Figura 2.12 (a)) ou no laço mais interno (Figura 2.12 (b)), o número de pontos de junção será o mesmo.

Algoritmo 1: juncaoPorAbrangencia(R,S,raio) Adaptado de (SERAPHIM, 2005)

```
1 para cada tupla r ∈ R faça
2     para cada tupla s ∈ S faça
3         se distancia(r,s) ≤ raio então
4             adiciona par ⟨r, s⟩ na lista de resultados
```

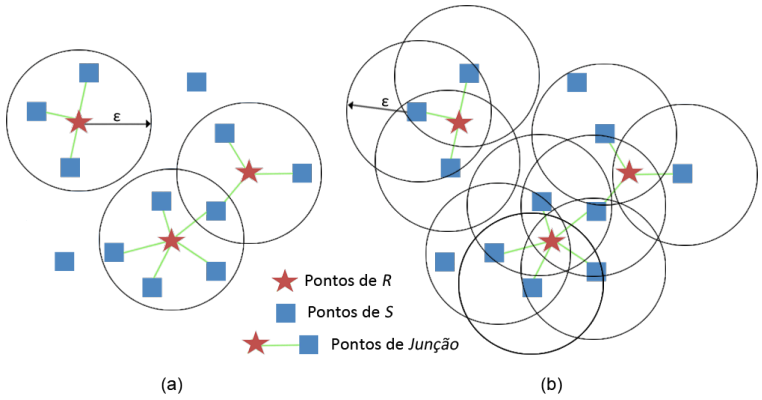


Figura 2.12: Junção por Abrangência e seu processo de comutatividade.

2.5.2 Junção por Vizinhos mais Próximos (*kNN-Join*)

Do mesmo jeito que a junção por abrangência, esta junção recebe como parâmetros de entrada dois conjuntos de dados R e S pertencentes a um mesmo domínio, porém, com um novo parâmetro adicional, uma quantidade k qualquer. O predicado desta junção busca os k pares de objetos mais próximos do conjunto R para cada objeto do conjunto S. Formalmente, esta junção é definida por $R \underset{k-nn}{\ltimes} S$, que é o menor subconjunto de $R \times S$ que contém para cada ponto de R no mínimo k pontos de S e para os quais obedece à seguinte condição:

$$\forall(r,s) \in R \underset{k-nn}{\ltimes} S, \forall(r,s') \in R \times S \vee R \underset{k-nn}{\ltimes} S : dist(r,s) < dist(r,s')$$

Caso haja empates entre os elementos obtidos nesta junção, é comum utilizar-se de duas abordagens: a primeira limita os elementos aos k elementos que satisfazem a regra ou então cria-se uma lista de empates. O Algoritmo 2 exemplifica como esta junção também pode ser implementada com o uso

de *loops* aninhados. Diferente da implementação por *loops* da junção por abrangência, esta operação não é comutativa, logo, a ordem das relações altera o resultado dos pares encontrados. A Figura 2.13 ilustra um exemplo da não comutatividade desta junção. Como pode ser visto, os k vizinhos mais próximos de cada elemento do conjunto R (Figura 2.13(a)) é diferente dos k vizinhos mais próximos de cada elemento do conjunto S (Figura 2.13(b)).

Algoritmo 2: kVizinhosMaisProximos(R,S,k,empate) Adaptado de (SERAPHIM, 2005)

```
1  Cria-se uma lista de resultados result com o tamanho de k
2  Cria-se uma lista de resultados auxiliar resultAux com o tamanho de k
3  para cada tupla r_i ∈ R faça
4      para cada tupla s_j ∈ S faça
5          se o tamanho de resultAux ≤ k então
6              adiciona par ⟨r_i, s_j⟩ em resultAux
7          senão
8              se distancia(r_i, s_j) ≤ maior distância ∈ resultAux então
9                  adiciona par ⟨r_i, s_j⟩ em resultAux
10                 remove de resultAux elementos excedentes à k verificando empate
11     adiciona resultAux em result
```

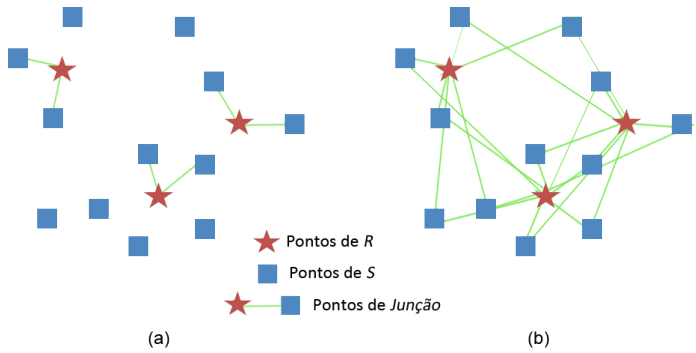


Figura 2.13: Junção por Vizinhos mais Próximos com k = 2 e a não comutatividade desta junção.

2.5.3 Junção por Proximidade (*kD-Join*)

Também do mesmo jeito que a Junção por Abrangência, esta junção recebe como parâmetros de entrada dois conjuntos de dados R e S pertencentes a um mesmo domínio, porém, o predicado desta junção recupera todos os k pares de $R \times S$ que possuem a menor distância entre os elementos de cada par. Formalmente esta junção é definida por $R \underset{k-CN}{\bowtie} S$ que é o menor subconjunto de $R \times S$ que contém no mínimo k pares de pontos e para os quais obedece à seguinte condição:

$$\forall(r,s) \in R \underset{k-CN}{\bowtie} S, \forall(r',s') \in R \times S \vee R \underset{k-CN}{\bowtie} S : dist(r,s) < dist(r',s')$$

O Algoritmo 3 exemplifica como esta junção também pode ser implementada com o uso de loops aninhados. Vale ressaltar que, do mesmo jeito que a Junção por Abrangência, esta junção também é uma operação comutativa, como ilustra a Figura 2.14. Como pode ser observado, os k pares de objetos mais próximos da junção $R \underset{k-CN}{\bowtie} S$ (Figura 2.14 (a)) são os mesmos que os k pares da junção $S \underset{k-CN}{\bowtie} R$ (Figura 2.14 (b)).

Algoritmo 3: juncaoPorProximidade(R,S,k,empate) Adaptado de (SERAPHIM, 2005)

```
1 Cria-se uma lista de resultados result com o tamanho de k
2 para cada tupla r_i ∈ R faça
3     para cada tupla s_j ∈ S faça
4         se o tamanho de result < k então
5             adiciona par ⟨r_i, s_j⟩ em result
6         senão
7             se distancia(r_i, s_j) ≤ maior distância ∈ result então
8                 adiciona par ⟨r_i, s_j⟩ em result
9                 remove de result elementos excedentes à k verificando empate
```

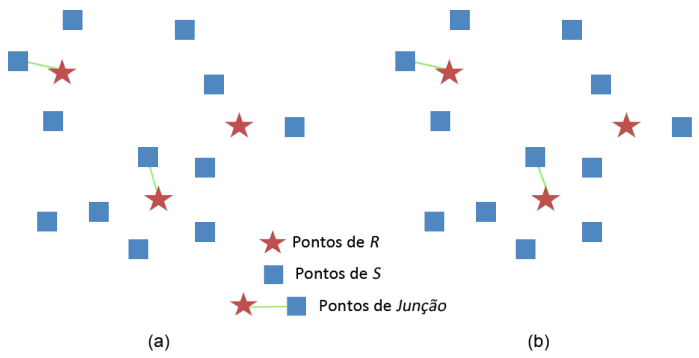


Figura 2.14: Junção por Proximidade com k = 2 e seu processo de comutatividade.

2.5.4 Junção ao Redor (*A-Join*)

Diferente das outras junções por similaridade que realizam operações específicas, esta junção combina propriedades da Junção por Vizinhos mais Próximos com propriedades da Junção por Abrangência de forma conjunta em uma única condição de junção. Esta junção também recebe como parâmetros de entrada dois conjuntos de dados R e S pertencentes a um mesmo domínio, mas o seu predicado recupera todos os k pares de $R \times S$ que possuem a menor distância entre os elementos de cada par e também que estejam a uma distância máxima ε uns dos outros. Formalmente, esta junção

é definida por $R \underset{A,MD=2\pi}{\bowtie} S$ que é o menor subconjunto de $R \times S$ que contém para cada ponto de R no mínimo k pares de pontos de S, que estejam a no máximo r de distância e para os quais obedece à seguinte condição:

$$\forall(r,s) \in R \underset{A,MD=2\pi}{\bowtie} S, \forall(r',s') \in R \times S \vee R \underset{A,MD=2\pi}{\bowtie} S : dist(r,s) < dist(r',s') \wedge dist(r,s) \leq r$$

O Algoritmo 4 exemplifica como esta junção também pode ser implementada com o uso de loops aninhados. Mais uma vez, vale ressaltar que do mesmo jeito que acontece na Junção por Abrangência e na Junção por Proximidade, esta junção também oferece comutatividade, como ilustra a Figura 2.15. Como pode ser visto, os pontos de junção de $R \underset{A,MD=2\pi}{\bowtie} S$, ilustrados na Figura 2.15 (a) e aqui representados com bordas pretas, também são os pontos de junção de $S \underset{A,MD=2\pi}{\bowtie} R$, como ilustrado pela Figura 2.15 (b).

Algoritmo 4: juncaoAoRedor(R,S,k,empate,raio) (DUARTE, 2012)

```
1  Cria-se uma lista de resultados result com o tamanho de k
2  Cria-se uma lista de resultados auxiliar resultAux com o tamanho de k
3  para cada tupla r_i ∈ R faça
4      para cada tupla s_j ∈ S faça
5          se o tamanho de resultAux ≤ k então
6              adiciona par ⟨r_i, s_j⟩ em resultAux
7          senão
8              se distancia(r_i, s_j) ≤ maior distância ∈ resultAux então
9                  adiciona par ⟨r_i, s_j⟩ em resultAux
10                 remove de resultAux elementos excedentes à k verificando empate
11 para cada tupla res_n ∈ resultAux faça
12     se distancia(⟨r_i, s_j⟩ ∈ res_n) ≤ raio então
13         adiciona res_n em result
```

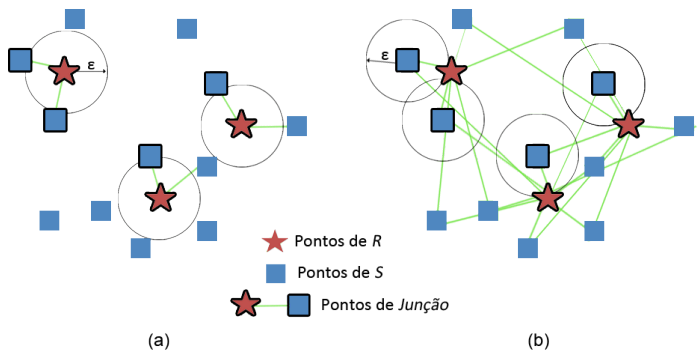


Figura 2.15: Junção ao Redor com k = 2, $r = 1$, e seu processo de comutatividade.

2.6 Junção Canalizada (*Channeled-Join*)

O operador denominado Junção Canalizada é um novo operador de junção por similaridade e principal objeto de estudo deste trabalho. Proposto por Duarte (2012), esta junção pode ser largamente utilizada em situações que envolvam operações sobre rotas em espaços métricos. Utilizando dela, perguntas do tipo: "Dado um caminho por onde será construído um oleoduto, mostrar os pontos de maior incidência de queimadas a menos de 5Km" ou "Quais são as cidades que estão até 16Km de distânca da rota formada pelas cidades de Pouso Alegre, Santa Rita do Sapucaí, Piranguinho e Itajubá/MG?" podem ser facilmente respondidas. A Figura 2.16 (a) ilustra este último exemplo.

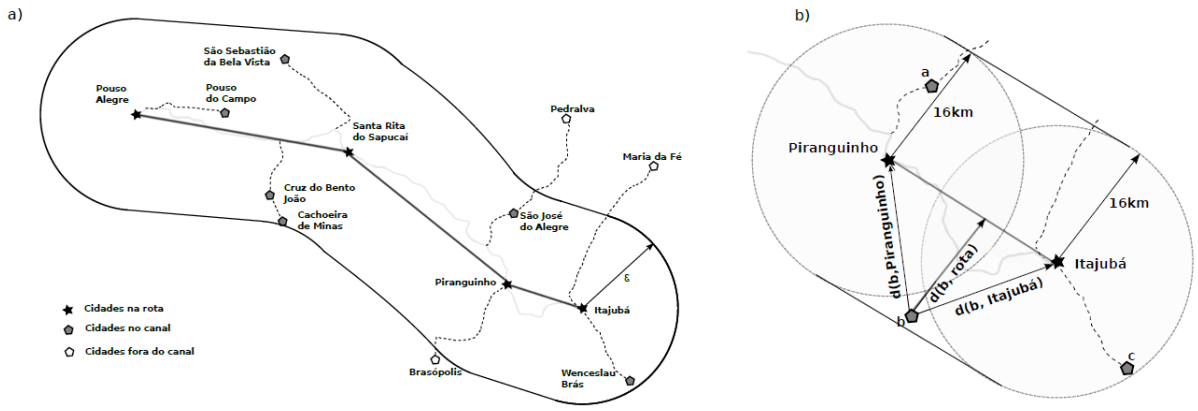


Figura 2.16: (a) Dimensões do Canal, (b) distância de um ponto a para a rota. Adaptada de (DUARTE, 2012)

A distância de 16Km para esta rota forma o canal que cobre as cidades que serão resposta para essa consulta: Pouso do Campo, São Sebastião da Bela Vista, Cruz do Bento João, Cachoeira de Minas, São José do Alegre e Wenceslau Braz. No conjunto de pontos que formam a base de dados, existem também pontos do domínio que não fazem parte da resposta por não estarem dentro do canal de 16Km como: Brasópolis, Pedralva e Maria da Fé.

A Junção Canalizada recebe como parâmetros de entrada dois conjuntos de dados R e S e uma distância máxima ε. A diferença é que um dos conjuntos representa uma rota de pontos que forma um grafo conexo acíclico de grau máximo dois.

Definindo que o conjunto S é formado por n pontos de interesse distintos, como por exemplo, as latitudes e longitudes das cidades brasileiras e o conjunto R é formado pelo conjunto de pontos que formam uma rota, por exemplo, latitudes e longitudes das cidades de Pouso Alegre, Santa Rita do Sapucaí, Piranguinho e Itajubá, a junção deve encontrar todos os pontos de interesse S que estejam a uma distância ε máxima da rota formada por pontos de R. A estratégia para responder a essa operação de junção é verificar se o ponto no domínio atende uma das duas situações de qualificação, que são:

- a distância até um dos pontos que formam a rota é menor que a distância máxima ε ou;

- se a menor distância para a reta que forma a rota é menor que a distância máxima ε.

A Figura 2.16 (b) ilustra essas situações de qualificação: uma parcela da rota, ou seja, Piranguinho e Itajubá, está representada por estrelas; os pontos do domínio *a,b* e *c* estão representados por pentágonos e o raio do canal é de 16Km. Na primeira estratégia de qualificação existem cidades do domínio (pentágonos) que estarão a menos de 16Km de pelo menos uma cidade da rota (estrela), logo, esta cidade do domínio fará parte do conjunto resposta. Como pode ser visto, o ponto *a* tem distância menor que 16Km em relação ao ponto Piranguinho na rota e o ponto *c* tem distância menor do que 16Km em relação ao ponto Itajubá na rota. No entanto, existem cidades do domínio, como o ponto *b*, cuja distância para as cidades Piranguinho e Itajubá na rota é maior do que 16Km, mas a sua distância para a rota formada entre estas cidades é inferior a 16Km. Neste caso, é necessário utilizar da geometria sobre a qual estão sujeitos os pontos para medir a menor distância entre o ponto do domínio e a rota existente entre os outros dois pontos. Detalhes de operação desta junção sobre a geometria euclidiana são discutidos no Capítulo 3, juntamente com propostas de novas estratégias que buscam explorar diferentes formas de executá-la.

2.7 Trabalhos Relacionados

Apesar da Junção Canalizada ser uma ideia relativamente nova e nenhum outro estudo até o momento ter se preocupado em explorá-la mais em detalhes, existem trabalhos e estudos que buscaram apresentar soluções e utilizar de recursos bem parecidos com os que são utilizados nela.

Xuan et al. (2008) em seu trabalho que lida com buscas por abrangência contínuas apresenta uma solução que responde à seguinte pergunta: "Dada uma rota, encontre todos os objetos de interesse (ex: hotéis) dentro de uma cobertura ε ao longo do caminho e retorne seus respectivos segmentos em que alguns destes objetos são acessíveis". Antes mesmo de explicar a solução de Xuan et al. (2008), é interessante ressaltar o conceito de buscas por abrangências contínuas, que muitas vezes podem ser confundidas com junção, por serem formas de buscas de dados bem parecidas. Segundo Wang, Zimmermann e Ku (2006), a busca por abrangência contínua é a busca de informação de objetos em movimento dentro de uma região definida pelo usuário e que continuamente monitora a mudança dos seus resultados dentro dela. Já a junção, como foi explicado anteriormente, envolve dois conjuntos de dados, para que seja realizado um produto cartesiano deles seguido por uma operação de seleção. Apesar de Xuan et al. (2008) não ter utilizado junção, sua solução é descrita da seguinte forma:

- Cada ponto da rota pode ser visto como o centro de círculos com raio ε;

- Encontre todos os objetos de interesse que estão dentro do raio de alcance dos pontos da rota e indexe os resultados em uma *R-Tree* (*Qpre-result()*);

- Transforme cada objeto de *Qpre-result()* no centro de novos círculos também com raio de alcance ε;

- Defina a interseção de pontos destes novos círculos com a rota como *split nodes*;

Como pode ser visto, ao invés da junção, Xuan et al. (2008) resolveu aplicar uma busca por abrangência estática em cada um dos pontos da rota a fim de coletar todos os objetos do domínio que se encontram dentro do raio de cobertura dela.

Já Song e Roussopoulos (2001) em seu trabalho também utilizou de uma estratégia bem parecida com a de Xuan et al. (2008), porém, vale ressaltar que ambas as estratégias apesar de serem válidas, podem sofrer de problemas de precisão e principalmente de desempenho, por conta da grande quantidade de pontos que uma determinada rota pode possuir. Os *split nodes*, citados na última etapa da solução de Xuan et al. (2008), nada mais são que pontos da rota que indicam quando os resultados da busca irão se modificar conforme o usuário caminha por ela (característica implícita da busca por abrangência contínua).

Assim como Xuan et al. (2008), Kalashnikov e Prabhakar (2003) também propôs soluções para serem aplicadas em buscas por abrangência contínuas, porém, utilizando de um operador de junção chamado *Grid-Join*, desenvolvido por ele e que opera sob um conjunto de dados espaciais. Assim como qualquer outra junção, este operador opera sobre dois conjuntos de pontos: um conjunto que possui pontos do domínio *(A)* e outro que é tratado como uma coleção de pontos que possuem círculos de raio ε *(B)* que são indexados usando de alguma estrutura de dados espacial. Esta junção é feita verificando se cada ponto do conjunto *A* se encontra dentro de cada um dos círculos indexados em *B*. Segundo Kalashnikov e Prabhakar (2003), uma vantagem desta abordagem (em oposição ao método tradicional de indexar os pontos de um conjunto e realizar buscas por abrangências estáticas sobre cada um dos pontos do outro conjunto) é que buscas pontuais são mais simples do que buscas por abrangência e, portanto, tendem a ser mais rápidas.

Também segundo Kalashnikov e Prabhakar (2003), é importante que a estrutura de dados que indexará os pontos com círculos (neste exemplo, o conjunto *B*) seja uma estrutura rápida. Por conta disso, testes realizados em Kalashnikov, Prabhakar e Hambrusch (2002) identificaram que em vez do uso de estruturas clássicas como *R-Tree* e suas variantes (*R*-Tree* e *CR-Tree*) serem utilizadas, uma nova variante da estrutura de dados denominada *Grid-index* foi utilizada para obter-se melhores resultados.

Outros exemplos de autores que também buscaram criar soluções parecidas com as de Xuan et al. (2008) e Kalashnikov e Prabhakar (2003) são: Yuan e Schneider (2010), que também utilizou de buscas por abrangência contínuas mas aplicadas em ambientes fechados como *shopping centers* e grandes construções, e Shekhar e Yoo (2003) que propôs quatro técnicas diferentes para se resolver um problema chamado de IRNN (*In-route Nearest Neighbour*, ou seja, vizinhos mais próximos de uma rota). Dentre as ténicas propostas, uma delas utiliza de uma junção entre dois conjuntos de dados (pontos do domínio e pontos da rota) indexados para localização dos n tipos de localidades (como postos de gasolina, bancos etc.) mais próximas de uma dada rota.

Assim como Shekhar e Yoo (2003), Deng et al. (2011) apresentou um problema semelhante que utiliza o conceito de trajetória para encontrar pontos mais próximos a um segmento de reta. O algoritmo proposto torna possível encontrar "todos os postos de gasolina mais próximos ao longo de um segmento de trajetória específico entre dois pontos". Esta consulta pode ser respondida usando-se de um tipo de consulta por trajetória chamada *P-Query*, que procura por pontos de interesse (POI - *Points of Interest*) que satisfazem um relacionamento espaço temporal para um ponto específico em um dado conjunto de pontos. Embora existam semelhanças no tipo de questão a ser respondida, o trabalho de Deng et al. (2011) foca na aplicação desta técnica na busca contínua por vizinhos mais próximos, assim como Shekhar e Yoo (2003).

Apesar dos trabalhos apresentados resolverem problemas parecidos, ou seja, todos visam buscar objetos de interesse ao longo de rotas, vale ressaltar também a importância das junções por similaridade, pois muitas vezes são operações essenciais de serem aplicadas na busca destes objetos. Existem vários outros trabalhos na literatura que utilizaram das primitivas das junções por similaridade para serem aplicadas em diversas outras soluções. De forma geral, estes trabalhos podem ser divididos em dois tipos: os que utilizam e os que não utilizam de estruturas para indexação dos conjuntos de dados envolvidos na junção.

Como exemplo de algoritmos e técnicas que utilizam de estruturas para indexação e que serviram de inspiração para este trabalho, pode-se citar:

- a solução de Brinkhoff, Kriegel e Seeger (1993) que indexa cada um dos conjuntos de dados em uma *R-Tree* e então realiza o cruzamento de seus índices para encontrar um conjunto de pares de objetos que casam um com o outro;

- a solução de Dohnal, Gennaro e Zezula (2003) que utiliza de uma estrutura de indexação customizada denominada *eD-index* que de forma contínua reparte os dados em partições menores a fim de realizar menores comparações e retornar respostas de junções mais rapidamente;

- Paredes e Reyes (2008) com sua solução denominada *List of Twin Clusters* que procura indexar

ambos os conjuntos de uma só vez em uma mesma estrutura de dados (ao invés da estratégia tradicional de indexar um ou os dois conjuntos independentemente) a fim de aumentar o desempenho das buscas;

- Lian e Chen (2010) que também utiliza da *M-Tree* como estrutura de indexação para dar suporte a uma operação denominada *Probabilistic Set Similarity Join (PS2J)* capaz de identificar documentos repetidos, errados e irrelevantes por meio de um limiar de similaridade entre eles;

- Zhou e Chen (2013) que utiliza de uma estrutura de indexação denominada *Variable Dimensional Extensible Hash (VDEH)* que visa armazenar mensagens provenientes da rede social *Twitter* para que constantemente seja capaz de aplicar junções por similaridade sobre elas, assim, identificando a ocorrência de novos eventos sociais automaticamente para facilitar a tomada de decisões de autoridades e equipes de resgate;

- Zhuang et al. (2015) faz uso de uma *B+-Tree* em uma de suas abordagens para armazenar características visuais, espaciais e textuais de imagens a fim de agilizar a execução de um operador de junção por similaridade chamado *Arbitrary Featured-based social image Similarity Join (AFS-Join)*, capaz de recuperar pares de imagens de diferentes usuários que são visual, espacial e textualmente parecidas;

Já em relação aos trabalhos que utilizam de junções por similaridade mas que não utilizam de estruturas de indexação para indexar os conjuntos de dados envolvidos pode-se citar como exemplo o *Epsilon Grid Order* (BöHM et al., 2001), *GESS (Generic External Space Sweep)* (DITTRICH; SEEGER, 2001), EGO*-Join (KALASHNIKOV; PRABHAKAR, 2007), Quick Join (JACOX; SAMET, 2008) e Super-EGO (KALASHNIKOV, 2013).

2.8 Considerações Finais

Neste Capítulo foram apresentados não só detalhes de operação e funcionamento de recursos que foram indispensáveis para a realização deste trabalho, mas também a revisão de trabalhos conhecidos da literatura que buscaram desenvolver soluções similares relacionadas à busca de dados.

Inicialmente, foram apresentados conceitos fundamentais a respeito de dados métricos e espaciais, juntamente com duas estruturas de dados consagradas utilizadas para indexação destes dados, neste caso, as estruturas de dados *M-Tree* e *R-Tree*, respectivamente. Como pôde ser visto, de forma geral, dados métricos são comparados e indexados se baseando em uma função de dissimilaridade que obedece às propriedades de simetria, não negatividade e desigualdade triangular e que mede o quanto os elementos são diferentes uns dos outros. Já para os dados espaciais, o processo de comparação e indexação acontece se baseando nos espaços ocupados pelos dados e na sobreposição deles.

Em seguida, detalhes de funcionamento do *Framework Object-Injection* foram apresentados. Como pôde ser visto, uma vez que este *framework* esteja acoplado a uma aplicação, é possível oferecer ao programador a persistência de objetos de forma nativa, para que o mesmo se concentre na lógica do seu problema e não nos detalhes de como os objetos podem ser armazenados.

Conceitos importantes relacionados às junções também foram apresentados, porém, com um foco mais voltado às junções por similaridade. Além dos detalhes de operação de quatro tipos de junções por similaridade bastante conhecidas terem sido apresentados, a introdução ao operador denominado Junção Canalizada também foi abordada. Como pôde ser visto, de forma geral, esta junção tem como principal objetivo buscar objetos de interesse que se encontram dentro do canal formado por uma rota.

Por fim, foram apresentados trabalhos consagrados da literatura que buscaram utilizar de recursos similares ao desse trabalho para a construção de soluções similares, ou seja, soluções que trabalham com a busca de dados métricos e espaciais.

No próximo capítulo serão apresentados não só maiores detalhes de operação da junção canalizada, mas também a formalização deste operador juntamente com a proposta de novas soluções que buscam aprimorá-lo.

Capítulo

3

Aprimoramentos da Junção Canalizada aplicada em dados Métricos e Espaciais

3.1 Considerações Iniciais

Este capítulo apresenta em detalhes as principais contribuições deste trabalho. A seção 3.2 apresenta a formalização do operador de junção canalizada por meio do cálculo relacional e a seção 3.3 descreve em detalhes não só o algoritmo básico da junção canalizada (implementação sequencial), mas também cada uma das propostas de aprimoramentos que podem ser feitos sobre ele quando aplicado em dados métricos e espaciais.

3.2 Definição da Junção Canalizada ($\otimes$)

A junção canalizada opera sobre dois conjuntos de dados R e S e uma distância máxima ε, em que R representa elementos que formam uma rota, ou seja, um grafo conexo acíclico de grau máximo dois, e S representa elementos de outro conjunto, neste caso, os pontos de interesse. A Figura 3.1 ilustra um exemplo da junção canalizada operando sobre os conjuntos $R = \{r1, r2\}$ e $S = \{s1, s2, s3\}$ e do canal formado pela distância máxima ε capaz de qualificar pontos de interesse. Aqui, dado dois pontos sequenciais da rota ($r1$ e $r2$), tem-se uma reta a formada pela distância entre eles. O espaço pode ser dividido em três divisões traçando duas retas paralelas entre si e perpendicular a reta a, uma passando pelo ponto $r1$ e outra passando por $r2$.

O elemento de estudo deste trabalho dentro da geometria euclidiana (WOLFE, 1945) é o triângulo e as relações existentes entre seus lados e ângulos. As seguintes relações extraídas da divisão do espaço euclidiano permitem estabelecer em que divisão pertence um determinado ponto de interesse:

$$D1 := \alpha \geq \pi/2 \leftrightarrow c^2 \geq a^2 + b^2$$

$$D3 := \beta \geq \pi/2 \leftrightarrow b^2 \geq a^2 + c^2$$

$$D2 := \neg D1 \wedge \neg D3$$

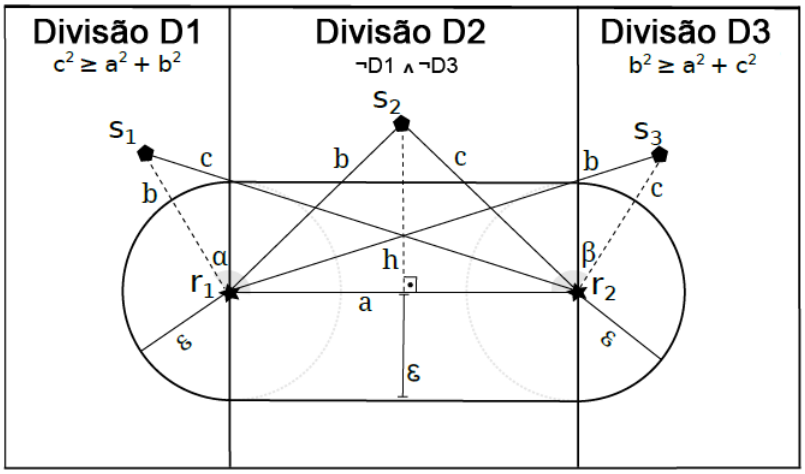


Figura 3.1: Regiões da Junção Canalizada. Adaptado de (DUARTE, 2012)

Pode-se determinar que um ponto de interesse $s1$ se encontra na *Divisão 1*, quando houver um ângulo obtuso α entre as retas formadas a partir das distâncias: de $r1$ para $r2$ (ambos pontos da rota) e de $s1$ (ponto de interesse) para $r1$ (ponto na rota). De forma similar, pode-se determinar que um ponto de interesse qualquer $s3$ se encontre na *Divisão 3*, quando houver um ângulo obtuso (β) entre as retas formadas a partir das distâncias: de $r1$ para $r2$ (ambos pontos da rota) e de $r2$ (ponto da rota) para $s3$ (ponto de interesse). Pela lei dos cossenos, um ângulo obtuso é formado em um triângulo quando o quadrado do cateto oposto ao ângulo é maior que a soma dos quadrados dos catetos adjacentes ao ângulo.

Por fim, para determinar que o ponto de interesse $s2$ seja resposta da consulta na *Divisão 2*, deve-se usar relações geométricas euclidianas para determinar a menor distância do ponto de interesse para a reta formada pela distância entre os pontos $r1$ e $r2$ na rota.

O símbolo $\overset{\varepsilon}{\bowtie}$ foi criado exclusivamente para representar a junção canalizada, sendo que a elipse que circunda o símbolo formal de junção (representado por $\bowtie$) refere-se ao canal que é formado pelos pontos da rota na junção canalizada. Formalmente, esta junção pode ser definida da seguinte forma:

$$R \overset{\varepsilon}{\bowtie} S$$

Para determinar que um ponto s_j qualquer do conjunto S esteja a uma distância ε de dois pontos

sequenciais r_i e r_{i+1} do conjunto R, ele deve obedecer à seguinte fórmula:

$$(r_i, s_j) \in R \times S \mid (D1 \wedge (b \leq \varepsilon)) \vee (D2 \wedge (h \leq \varepsilon)) \vee (D3 \wedge (c \leq \varepsilon))$$

onde,

$$a := dist(r_i, r_{i+1})$$

$$b := dist(r_i, s_j)$$

$$c := dist(r_{i+1}, s_j)$$

$$D1 := c^2 \geq a^2 + b^2,$$

$$D3 := b^2 \geq a^2 + c^2,$$

$$D2 := \neg D1 \wedge \neg D3$$

$$h = \frac{2A}{a}$$

$$A = \sqrt{P(P-a)(P-b)(P-c)}$$

$$P = \frac{a+b+c}{2}$$

Resumidamente, para ser resposta da junção canalizada, um ponto de interesse s_j do conjunto S deve:

- estar na Divisão D1 e a distância b deve ser menor ou igual ao canal ε. A reta b é formada pela distância entre o ponto de interesse s_j e o ponto r_i da rota localizado no conjunto R;
- estar na Divisão D3 e a distância c deve ser menor ou igual ao canal ε. A reta c é formada pela distância entre o ponto de interesse s_j e o ponto r_{i+1} da rota localizado no conjunto R;
- estar na divisão D2 e a altura h ser menor ou igual ao canal ε. A altura h é perpendicular à reta a e é a menor distância do ponto de interesse s_j para a reta a, que é formada por dois pontos sequenciais r_i e r_{i+1} pertencentes à rota. Para determinar a altura h, primeiramente é necessário

obter a área A do triângulo que pode ser obtida pelo Teorema de Heron (definições A e P da fórmula).

3.3 Algoritmos para a Junção Canalizada

O Algoritmo 5 (**Junção Canalizada Sequencial**), definido no trabalho de Duarte (2012), foi a primeira proposta de algoritmo desenvolvida para execução da junção canalizada, juntamente com outra proposta que faz uso da estrutura de dados *Slim-Tree* (TRAINA et al., 2002). Com exceção do algoritmo sequencial, todos os outros algoritmos aqui apresentados fizeram uso das estrutudas de dados *M-Tree* e *R-Tree*, e são contribuições deste trabalho.

Algoritmo 5: jCSequencial(S,R,ε)

```
1  para cada bloco b_k ∈ arquivo sequencial S faça
2      para cada ponto s_j ∈ b_k faça
3          para cada ponto r_i ∈ R faça
4              a := dist(r_i, r_{i+1})
5              b := dist(r_i, s_j)
6              c := dist(r_{i+1}, s_j)
7              se (c^2 ≥ a^2 + b^2) ∧ (b ≤ ε) então
8                  adiciona s_j ao conjunto resposta e interrompa
9              senão
10                 se (b^2 ≥ a^2 + c^2) ∧ (c ≤ ε) então
11                     adiciona s_j ao conjunto resposta e interrompa
12                 senão
13                     P = (a + b + c)/2
14                     A = √(P(P − a)(P − b)(P − c))
15                     h = 2A/a
16                     se h ≤ ε então
17                         adiciona s_j ao conjunto resposta e interrompa
```

Este algoritmo trabalha com a estratégia de *loops* aninhados, e começa sua execução fazendo leitura dos blocos pelos quais estão armazenados os pontos de interesse (linha 1). A cada novo bloco lido, a junção canalizada vai sendo aplicada, buscando encontrar elementos que satisfaçam suas condições. Primeiramente, acontece o produto cartesiano entre dois conjuntos de dados (representados pelos laços nas linhas 2 e 3): no caso, os pontos de interesse (localizados dentro do conjunto *S*) e os pontos que formam a rota (localizados no conjunto *R*). Em seguida, acontece o processo de seleção. Primeiramente, o algoritmo busca verificar a divisão pelo qual um ponto de interesse pertence se baseando em um canal formado a cada dois pontos da rota, representados por r_i, r_{i+1}. Caso o ponto de

interesse se encontre nas divisões 1 ou 3 e esteja dentro do canal formado por ε (linhas 7 e 10), ele é adicionado ao conjunto resposta (linhas 8 e 11) ou caso o ponto de interesse se encontre dentro da divisão 2 e sua altura em relação à linha formada por r_k, r_{k+1} esteja dentro do canal formado por ε (linha 16), ele também é adicionado ao conjunto resposta (linha 17).

Outro detalhe importante da implementação sequencial é que, dentre todos os algoritmos propostos para execução da junção canalizada, este se apresenta como o único que permite a comutatividade dos conjuntos de dados envolvidos. Também é importante ressaltar que, apesar da implementação deste algoritmo ser simples, ela pode sofrer de graves problemas de desempenho, pois é necessário percorrer todos os blocos do arquivo sequencial, verificando a divisão que se encontra cada ponto s_j do conjunto S e também se cada um deles se encontra dentro do canal formado pelos pontos r_i e r_{i+1} da rota.

Para diminuir a quantidade de pontos a serem analisados, e consequentemente, diminuir cálculos desnecessários, estruturas de dados do tipo árvore como a *M-Tree* (aplicada em dados métricos) e *R-Tree* (aplicada em dados espaciais) podem ser utilizadas, pois trabalham com o conceito de agrupamento de dados como bolas e hiper-retângulos, respectivamente. A Figura 3.2 ilustra um exemplo de como os blocos índices e folhas da *M-Tree* são organizados e como acontece a interação entre uma rota e os pontos de interesse quando se aplica a Junção Canalizada.

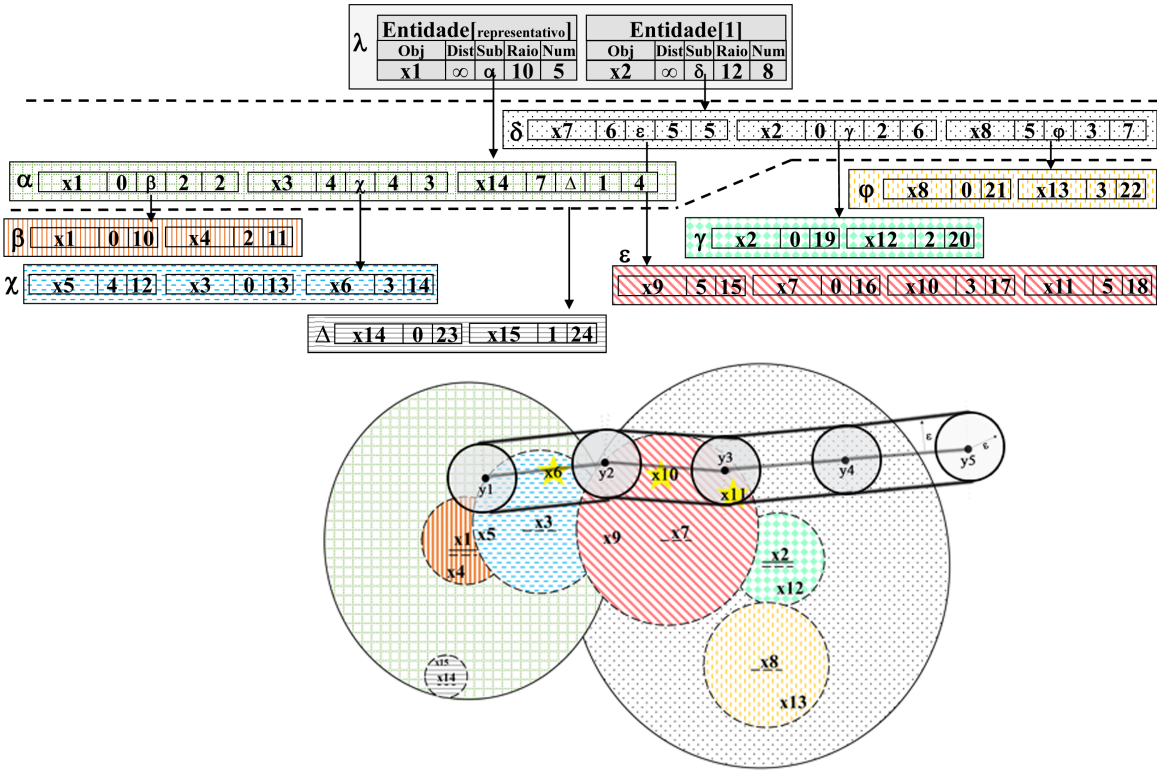


Figura 3.2: Organização dos blocos índices e folhas na *M-Tree* e interação entre rotas e pontos de interesse na Junção Canalizada.

Os pontos xn representam pontos de interesse organizados pela *M-Tree* após o processo de indexação, já os pontos yn representam pontos que formam uma rota. Como pode ser visto, os pontos da rota são capazes de formarem um canal capaz de qualificar pontos que se encontram dentro dos círculos construídos pela *M-Tree*. Caso algum destes círculos possua regiões que formem uma interseção com o canal formado pela rota, seus pontos ou círculos mais internos também precisam ser verificados, pois possuem chance de estarem dentro do canal e serem resposta da junção. Porém, caso não haja uma interseção entre um determinado círculo e a rota, tanto ele quanto seus círculos mais internos serão descartados e não precisarão ser verificados, poupando implicitamente verificações ou cálculos de distância desnecessários.

O Algoritmo 6, intitulado **Junção Canalizada na *M-Tree* com Primeira Qualificação**, foi a primeira proposta de implementação desta junção fazendo uso da estrutura de dados *M-Tree*. De forma geral, o algoritmo verifica se cada um dos blocos mais internos do $blocoM$, sejam eles índices ou folhas, possuem ou não interseção com o canal formado por cada par de pontos r_i e r_{i+1} da rota. Inicialmente, cada sub-bloco (s_j) do $blocoM$ é verificado (linha 1). Caso um sub-bloco seja do tipo índice, sua cobertura precisa ser guardada (linha 2), pois após determinada a região pelo qual se encontra, esta cobertura deve ser somada à largura do canal ε para verificar se ele possui ou não interseção com o canal formado pelos pontos da rota (linhas 7, 13 e 22). Caso exista interseção e este bloco seja do tipo índice, ele deve ser passado como argumento para a função que de forma recursiva verifica se seus blocos mais internos também possuem interseções (linhas 9, 15 e 24). Caso exista interseção e o bloco seja do tipo folha, ele simplesmente é adicionado ao conjunto resposta e o laço mais interno é interrompido, assim como acontece no algoritmo sequencial (linhas 11, 17 e 26).

Na Figura 3.2, por exemplo, os blocos que guardam os círculos x1,x3 e x7 precisam ser avaliados, pois possuem interseção com o canal, já os blocos x2, x8 e x14, não possuem. Uma vantagem clara do uso das implementações que usam a *M-Tree* em relação à implementação usando arquivo sequencial é a economia com cálculos de distância que pode acontecer caso blocos do tipo índice não sejam resposta, pois como são formados por outros sub-blocos que podem ser índices ou folhas, a não qualificação resulta em menos pontos de interesse a serem avaliados no decorrer da execução do algoritmo.

O Algoritmo 6 além de desfrutar das vantagens oferecidas pela estrutura de dados *M-Tree*, também é capaz de evitar ainda mais laços desnecessários, pois na primeira oportunidade que tem de qualificar um sub-bloco, já interrompe o laço mais interno, que é formado por pontos da rota (linhas 9, 11, 15, 17, 24 e 26). Por isso o nome "Primeira Qualificação".

O Algoritmo 7, denominado **Junção Canalizada na *M-Tree* com Primeira Qualificação e Posição da Rota**, tem uma operação bem parecida com a implementação do Algoritmo 6, porém, adiciona

um novo recurso que busca limitar o tamanho da rota que precisa ser analisada em um determinado momento. Neste algoritmo, toda vez que um sub-bloco índice é qualificado (linhas 8, 14 e 23), passa-se não só este sub-bloco para a função recursiva como também a posição da rota que o qualificou (linhas 9, 15 e 24). Esta posição é utilizada nas próximas execuções como início da rota para seus blocos mais internos (linha 3), evitando assim, que partes da rota que já foram avaliadas (não mais relevantes) sejam avaliadas novamente.

Algoritmo 6: jCMTreePrimeiraQualificacao($blocoM$,R,ε)

```
1  para cada elemento s_j ∈ blocoM faça
2      cov := (blocoM é índice) ? cobertura de s_j : 0
3      para cada elemento r_i ∈ R faça
4          a := dist(r_i, r_{i+1})
5          b := dist(r_i, s_j)
6          c := dist(r_{i+1}, s_j)
7          se (c^2 ≥ a^2 + b^2) ∧ (b ≤ cov + ε) então
8              se blocoM é índice então
9                  executa jCMTreePrimeiraQualificacao(sub_j,R,ε) e interrompa
10             senão
11                 adiciona s_j ao conjunto resposta e interrompa
12         senão
13             se (b^2 ≥ a^2 + c^2) ∧ (c ≤ cov + ε) então
14                 se blocoM é índice então
15                     executa jCMTreePrimeiraQualificacao(sub_j,R,ε) e interrompa
16                 senão
17                     adiciona s_j ao conjunto resposta e interrompa
18             senão
19                 P = (a + b + c)/2
20                 A = √(P(P − a)(P − b)(P − c))
21                 h = 2A/a
22                 se (h ≤ cov + ε) então
23                     se blocoM é índice então
24                         executa jCMTreePrimeiraQualificacao(sub_j,R,ε) e interrompa
25                     senão
26                         adiciona s_j ao conjunto resposta e interrompa
```

Já o Algoritmo 8, chamado de **Junção Canalizada na *M-Tree* com Qualificação Mínima**, utiliza de novas estratégias que também buscam diminuir cálculos de distância, e consequentemente, processamentos desnecessários. De forma geral, este algoritmo funciona da seguinte maneira: quando um

novo bloco s_j é carregado para ser analisado, toda a rota deve ser percorrida a fim de verificar qual é a menor distância que s_j possui com o canal, pois o bloco pode chegar a ser qualificado por diferentes regiões da rota. Esta distância mínima deve ser guardada para que seja possível aplicar a técnica de "poda" por desigualdade triangular nos sub-blocos deste bloco, permitindo verificar com antecedência se eles realmente precisarão ser analizados ou poderão ser "podados". Esta distância mínima é armazenada pela variável *min*, que vai sendo constantemente atualizada até que o laço interno finalize (linhas 12, 19 e 29).

Algoritmo 7: jCMTreePrimeiraQualificacaoPosicaoDaRota($blocoM$,R,ε,pos)

```
1  para cada elemento s_j ∈ blocoM faça
2      cov := (blocoM é índice) ? cobertura de s_j : 0
3      para cada elemento r_i ∈ R sendo i iniciado em pos faça
4          a := dist(r_i, r_{i+1})
5          b := dist(r_i, s_j)
6          c := dist(r_{i+1}, s_j)
7          se (c² ≥ a² + b²) ∧ (b ≤ cov + ε) então
8              se blocoM é índice então
9                  executa jCMTreePrimeiraQualificacaoPosicaoDaRota(sub_j,R,ε,i) e interrompa
10             senão
11                 adiciona s_j ao conjunto resposta e interrompa
12         senão
13             se (b² ≥ a² + c²) ∧ (c ≤ cov + ε) então
14                 se blocoM é índice então
15                     executa jCMTreePrimeiraQualificacaoPosicaoDaRota(sub_j,R,ε,i) e interrompa
16                 senão
17                     adiciona s_j ao conjunto resposta e interrompa
18             senão
19                 P = (a + b + c)/2
20                 A = √(P(P − a)(P − b)(P − c))
21                 h = 2A/a
22                 se (h ≤ cov + ε) então
23                     se blocoM é índice então
24                         executa jCMTreePrimeiraQualificacaoPosicaoDaRota(sub_j,R,ε,i) e interrompa
25                     senão
26                         adiciona s_j ao conjunto resposta e interrompa
```

Todo bloco antes de ser analisado mais em detalhes, seja ele índice ou folha, passa pela "poda" por desigualdade triangular, presente na linha 5.

Algoritmo 8: jCMTreeQualificaçãoMinima(*blocoM*,R,ε,minDRep)

```
1  para cada elemento s_j ∈ blocoM faça
2      jmin := 0
3      min := ∞
4      cov := (blocoM é índice) ? cobertura de s_j : 0
5      se minDRep + ε ≥ distRep(s_j) - cov então
6          para cada elemento r_i ∈ R faça
7              a := dist(r_i, r_{i+1})
8              b := dist(r_i, s_j)
9              c := dist(r_{i+1}, s_j)
10             se (c² ≥ a² + b²) ∧ (b ≤ cov + ε) ∧ (b ≤ min) então
11                 se blocoM é índice então
12                     min := b
13                     jmin := j
14                 senão
15                     adiciona s_j ao conjunto resposta e interrompa
16             senão
17                 se (b² ≥ a² + c²) ∧ (c ≤ cov + ε) ∧ (c ≤ min) então
18                     se blocoM é índice então
19                         min := c
20                         jmin := j
21                     senão
22                         adiciona s_j ao conjunto resposta e interrompa
23                 senão
24                     P = (a + b + c)/2
25                     A = √(P(P − a)(P − b)(P − c))
26                     h = 2A/a
27                     se (h ≤ cov + ε) então
28                         se blocoM é índice então
29                             min := 0
30                             jmin := j
31                         senão
32                             adiciona s_j ao conjunto resposta e interrompa
33         se blocoM é índice ∧ (min <> ∞) então
34             jCMTreeQualificaçãoMinima(sub_jmin,R,ε,min)
```

A fórmula da desigualdade triangular que indica a necessidade de analisar distâncias é:

$$\textit{minDRep} + \varepsilon \geq \textit{distRep}(s_j) - \textit{cov},$$

onde *minDRep* é a distância mínima do representativo (pai) do bloco a um ponto da rota que o qualificou anteriormente, ε é a largura do canal, *distRep*(s_j) é a distância do bloco ao seu representativo e *cov* é a cobertura do bloco. A Figura 3.3 ilustra por meio de um exemplo como a aplicação da "poda" por desigualdade triangular pode auxiliar na exclusão de blocos que mesmo se fossem processados, não seriam resposta, como é o caso do bloco *x14*.

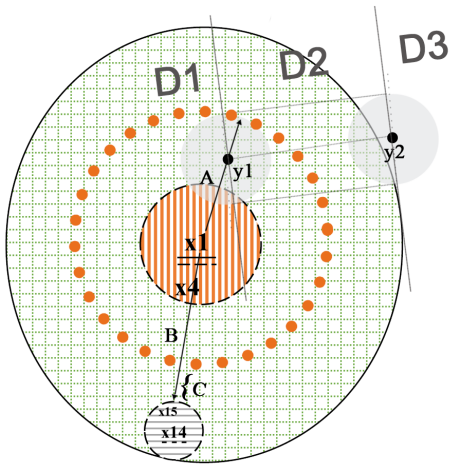


Figura 3.3: Exemplo de operação da poda por desigualdade triangular.

Como pode ser visto, após ser verificado por todos os pontos da rota, o ponto *y1* foi o ponto mais próximo responsável por qualificar o bloco *x1*. Como a distância deste bloco a este ponto da rota é guardada (*minDRep*), sub-blocos de *x1* poderão utilizar deste valor para verificarem se devem ou não ser "podados". A distância de *x1* ao ponto *y1* somada com a largura do canal ε formam uma região chamada de "área de aceitação" de *x1*. Se sub-blocos pertencentes a *x1* ao serem avaliados não se encontrarem dentro ou tangenciarem esta área, quer dizer que não possuem a mínima chance de serem resposta da consulta, por isso, acabam sendo "podados".

Na Figura 3.3, a seta A = *minDRep* + ε, a seta B = distRep(s_j) - *cov* e C é o espaço entre o círculo formado por A e a distância B. Como pode ser visto, a distância B não tangencia e nem se encontra dentro da área de aceitação de $x1$, por isso, *x14* acaba sendo "podado".

O Algoritmo 9, nomeado **Junção Canalizada na *M-Tree* com Qualificação Mínima e Posição da Rota**, foi o último algoritmo proposto neste trabalho que utilizou da estrutura de dados *M-Tree*. Ele utiliza de forma conjunta duas estratégias (linhas 7 e 38) que foram utilizadas nos algoritmos anteriores a fim de buscar diminuir ainda mais o número de cálculos de distância desnecessários. Neste caso, as estratégias utilizadas foram: o armazenamento da posição da rota que qualifica um bloco (que evita sub-blocos caminharem por posições da rotas que já foram avaliadas pelo seu bloco representativo) e a distância para essa posição da rota (que é utilizada para "podar" sub-blocos por meio da desigualdade triangular). A única diferença é que, neste algoritmo, a posição da rota que

qualifica um bloco é armazenada pela variável *imin* (linhas 13, 21 e 32).

Algoritmo 9: jCMTreeQualificaçãoMinimaPosicaoDaRota($blocoM$,R,ε,minDRep,pos)

```
1  para cada elemento s_j ∈ blocoM faça
2      imin := 0
3      jmin := 0
4      min := ∞
5      cov := (blocoM é índice) ? cobertura de s_j : 0
6      se minDRep + ε ≥ distRep(s_j) - cov então
7          para cada elemento r_i ∈ R sendo i iniciado em pos faça
8              a := dist(r_i, r_{i+1})
9              b := dist(r_i, s_j)
10             c := dist(r_{i+1}, s_j)
11             se (c² ≥ a² + b²) ∧ (b ≤ cov + ε) ∧ (b ≤ min) então
12                 se blocoM é índice então
13                     imin := i
14                     jmin := j
15                     min := b
16                 senão
17                     adiciona s_j ao conjunto resposta e interrompa
18             senão
19                 se (b² ≥ a² + c²) ∧ (c ≤ cov + ε) ∧ (c ≤ min) então
20                     se blocoM é índice então
21                         imin := i
22                         jmin := j
23                         min := c
24                     senão
25                         adiciona s_j ao conjunto resposta e interrompa
26                 senão
27                     P = (a + b + c)/2
28                     A = √(P(P − a)(P − b)(P − c))
29                     h = 2A/a
30                     se (h ≤ cov + ε) ∧ (h ≤ min) então
31                         se blocoM é índice então
32                             imin := i
33                             jmin := j
34                             min := 0
35                         senão
36                             adiciona s_j ao conjunto resposta e interrompa
37         se blocoM é índice ∧ (min <> ∞) então
38             jCMTreeQualificaçãoMinimaPosicaoDaRota(sub_jmin,R,ε,min,imin)
```

Por fim, o Algoritmo 10 (**Junção Canalizada na *R-Tree* com Primeira Qualificação e Posição da Rota**) foi a última proposta de implementação da Junção Canalizada neste trabalho, porém, foi a primeira proposta de implementação deste operador fazendo-se uso da estrutura de dados *R-Tree*. A Figura 3.4 ilustra um exemplo de como retângulos formados pelos pontos da rota interagem com os MBR's (*Minimum Bouding Rectangles*) formados pelos pontos de interesse.

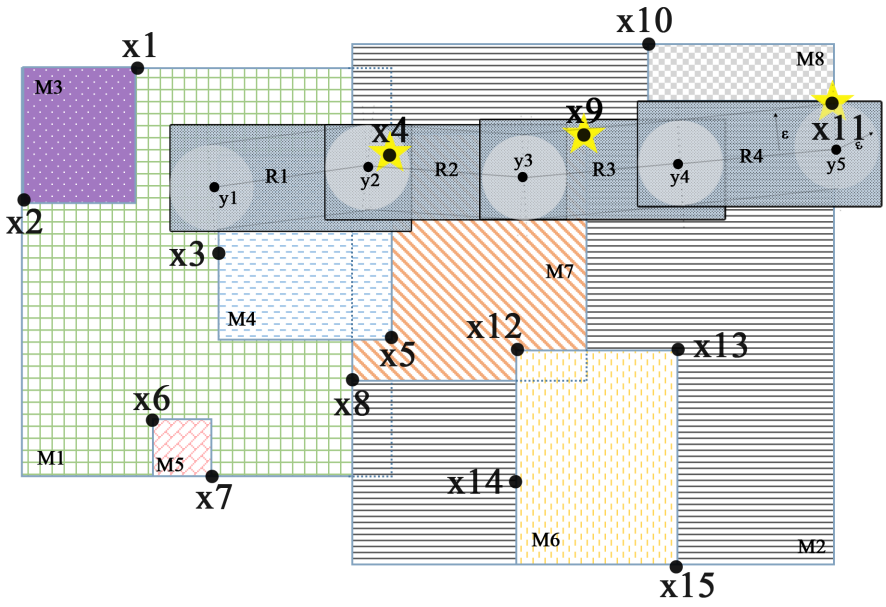


Figura 3.4: Interação entre rotas e pontos do domínio da Junção Canalizada na *R-Tree*.

Os retângulos Mn representam os MBR's formados pelos pontos de interesse xn após o processo de indexação. Os retângulos Rn representam os retângulos formados a cada dois pontos yn consecutivos da rota. Como pode ser visto, os pontos da rota são capazes de formarem um canal de retângulos capaz de qualificar MBR's construídos pela *R-Tree*. Caso algum destes MBR's possuam regiões de interseção com o canal formado pela rota, seus pontos ou MBR's mais internos também precisam ser verificados, pois possuem chance de estarem dentro do canal e serem resposta da junção. Porém, caso não haja uma interseção de um MBR específico com a rota, todos os MBR's e pontos mais internos pertencentes a ele serão descartados e não precisarão ser verificados, poupando implicitamente verificações ou cálculos de distância desnecessários, assim como acontece nas implementações baseadas na *M-Tree*.

Um detalhe importante que pode ser observado na análise de blocos índices deste algoritmo é a ausência dos cálculos de divisões pelo qual MBR's poderiam se encontrar. Aqui, os cálculos de divisões não podem ser feitos nestes blocos do mesmo jeito que foi feito nos blocos índices da *M-Tree*, pois a técnica proposta por Duarte (2012) não é ideal para se trabalhar com retângulos, mas apenas com pontos e círculos. Por conta disso, para se calcular a interseção entre os blocos formados pela

rota e os MBR's, utiliza-se da função *temSobreposicao()*, apresentada no Algoritmo 11. Essa função retorna *false* se não existir sobreposição entre dois MBR's (linhas 3 e 5), ou *true*, caso exista (linha 6).

Algoritmo 10: jCRTreePrimeiraQualificacaoPosicaoDaRota($blocoR$,R,ε,pos)

```
1  para cada mbr_j ∈ blocoR faça
2      para cada elemento r_i ∈ R sendo i iniciado em pos faça
3          se blocoR é índice então
4              mbrRota := uniao(r_i ,r_{i+1})
5              se temSobreposicao(mbr_j,mbrRota) então
6                  jCRTreePrimeiraQualificacaoPosicaoDaRota(sub_j,R,ε,i)
7          senão
8              a := dist(r_i, r_{i+1})
9              b := dist(r_i, mbr_j)
10             c := dist(r_{i+1}, mbr_j)
11             se (c² ≥ a² + b²) ∧ (b ≤ ε) então
12                 adiciona mbr_j ao conjunto resposta e interrompa
13             senão
14                 se (b² ≥ a² + c²) ∧ (c ≤ ε) então
15                     adiciona mbr_j ao conjunto resposta e interrompa
16                 senão
17                     P = (a + b + c)/2
18                     A = √(P(P − a)(P − b)(P − c))
19                     h = 2A/a
20                     se (h ≤ ε) então
21                         adiciona mbr_j ao conjunto resposta e interrompa
```

Apesar da abstração, uma das desvantagens claramente visíveis deste método é não permitir a realização de "podas" em blocos da árvore, diferente do que acontece nos algoritmos 8 e 9, que permitem. Já para blocos folha, o procedimento de avaliação acaba sendo praticamente o mesmo que acontece na implementação sequencial. A única diferença é que, neste caso, o laço mais externo é formado por blocos do tipo folha da *R-Tree*, ou seja, os blocos que guardam os pontos do domínio.

Algoritmo 11: temSobreposicao(mbr1,mbr2)

```
1  para i de 0 até dimensoes(mbr1) faça
2      se origem(mbr1,i) > origem(mbr2,i) + extensao(mbr2,i) então
3          retorna false
4      se origem(mbr1,i) + extensao(mbr1,i) < origem(mbr2,i) então
5          retorna false
6  retorna true
```

Capítulo
4
Experimentos e Resultados

Este Capítulo descreve 3 experimentos realizados para avaliar as diferentes implementações dos algoritmos propostos para execução da junção canalizada, tanto para dados métricos (implementações na *M-Tree*) quanto para dados espaciais (*R-Tree*). A seção 4.1 descreve os experimentos da junção canalizada utilizando de uma base de pontos uniformemente distribuída. Já a seção 4.2 apresenta a junção canalizada sendo executada sobre uma base de dados formada por posições geográficas dos códigos postais do Reino Unido. E por fim, a seção 4.3 apresenta experimentos que usam de imagens pertercentes à base denominada ALOI (*Amsterdam Library of Object Images*) (IMAGES, 2004).

Os experimentos foram realizados em um computador com processador Intel Core i5-5200U (com *clock* de 2.20GHz, 2 Núcleo(s), 4 Processadores lógicos), 8 GB de memória RAM, Sistema Operacional *Microsoft Windows 10 Pro 64bits* e HD SATA de 1 TB com 5400RPM. Para execução dos experimentos, que foram todos feitos na linguagem de programação Java, utilizou-se a JDK 1.8*release*112.

Cada um dos 3 experimentos utilizou de duas estruturas de dados: a estrutura com a extensão *.mtree*, que é utilizada pelos algoritmos que fazem uso da *M-Tree*, e a estrutura com a extensão *.rtree*, utilizada pelo algoritmo de junção canalizada que faz uso da *R-Tree*. Somente no experimento 3 (ALOI), utilizou-se da estrutura com a extensão *.seq* que é utilizada no algoritmo sequencial.

Todos os experimentos foram executados sobre estruturas de dados que utilizaram blocos em disco de tamanhos 1KB e 2KB. A tabela 4.1 indica qual foi o tempo, o número de verificações e a quantidade de acessos a disco que foram necessários para gerar cada uma das estruturas de dados que são utilizadas pelos algoritmos da junção canalizada (neste caso, as estruturas *.seq*, *.mtree* e *.rtree*). Ela mostra também a altura das árvores geradas, tanto para estruturas que fazem uso de blocos de tamanho 1KB quanto de 2KB.

Cada um dos experimentos trabalhou com uma base de pontos de interesse de tipos e tamanhos diferentes, que serão detalhadas nas próximas seções. Na Tabela 4.1 isso é claramente notável devido ao tempo que foi necessário para indexar cada uma destas bases nas estruturas de dados. De forma

Tabela 4.1: Informações relevantes sobre a construção das estruturas utilizadas pelos algoritmos da junção canalizada.

Experimento	Bloco	Estrutura	Tempo de Geração	Verificações	Acessos a Disco	Altura da Árvore
Uniforme	1KB	Sequencial	10,87 h	-	209.999.999	-
Uniforme	1KB	*M-Tree*	165,54 h	9.256.502.231	846.721.212	7
Uniforme	1KB	*R-Tree*	314,34 h	11.352.458.351	881.686.851	8
Uniforme	2KB	Sequencial	2,5 h	-	204.878.049	-
Uniforme	2KB	*M-Tree*	146,1 h	13.111.943.791	692.092.647	6
Uniforme	2KB	*R-Tree*	257,08 h	17.097.057.860	707.412.510	6
UK *Postcodes*	1KB	Sequencial	0,85 min	-	3.441.081	-
UK *Postcodes*	1KB	*M-Tree*	2,7 min	157.308.642	9.461.773	5
UK *Postcodes*	1KB	*R-Tree*	3.9 min	236.032.775	9.731.487	5
UK *Postcodes*	2KB	Sequencial	0,77 min	-	3.441.081	-
UK *Postcodes*	2KB	*M-Tree*	2,4 min	157.308.642	9.461.773	5
UK *Postcodes*	2KB	*R-Tree*	3,5 min	236.032.775	9.731.487	5
ALOI	1KB	Sequencial	3,9 s	-	264.599	-
ALOI	1KB	*M-Tree*	11,7 s	4.979.065	1.161.190	8
ALOI	1KB	*R-Tree*	20,4 s	3.866.969	1.634.427	11
ALOI	2KB	Sequencial	3,9 s	-	240.545	-
ALOI	2KB	*M-Tree*	9,8 s	5.219.551	729.054	6
ALOI	2KB	*R-Tree*	13,1 s	4.542.352	877.890	7

geral, pode-se dizer que o experimento UNIFORME precisou de horas para gerar suas estruturas, o *UK Postcodes* precisou de alguns minutos e o ALOI de alguns segundos.

Para construção das estruturas do tipo *.seq*, o número de verificações é nulo, pois não há necessidade de organização dos pontos do domínio de uma forma especial, exigindo que algum cálculo organizacional seja realizado. Já para construção das estruturas com extensão *.mtree*, as verificações refletem as quantidades de cálculos de distância que foram realizados para organização dos círculos que compõem as árvores. No caso das estruturas com extensão *.rtree*, as verificações refletem as quantidades de cálculos de sobreposição que foram necessários para formação dos MBR's.

Como também observa-se na Tabela 4.1, quanto maior o tamanho do bloco, maior ou igual será o número de verificações necessárias para geração das estruturas, uma vez que blocos maiores guardam mais objetos a serem verificados.

Para todas as estruturas, independente do seu tipo, o acesso a disco indica a quantidade de operações de leitura e escrita que foram realizadas na memória secundária durante a construção de cada uma delas. Observa-se também que, quanto maior o tamanho do bloco, menor ou igual tende a ser o número de acessos a disco, pois mais informações podem ser armazenadas em blocos maiores.

Por fim, a altura da árvore significa a profundidade da mesma, ou seja, o número máximo de níveis que foram formados na árvore após o processo de indexação. Mais uma vez, para a estrutura *.seq*, esta informação é nula, por conta das informações serem alocadas no arquivo de forma sequencial. Quanto maior o tamanho do bloco, menor ou igual tende a ser a árvore gerada, pois como blocos maiores são capazes de armazenar mais informações, menos blocos acabam sendo gerados no final da construção de uma árvore.

O resultado retornado pelos três experimentos foram as médias do tempo de execução, quanti-

dade de verificações nas estruturas de dados e acesso a disco das quantidades de junções sobre cada algoritmo proposto. Nos gráficos que serão apresentados nas seções 4.1, 4.2 e 4.3, os nomes dos algoritmos foram abreviados a fim de descrevê-los de forma mais prática, são eles:

- ***sequential***: Junção Canalizada Sequencial;
- ***firstQuali***: Junção Canalizada na *M-Tree* com Primeira Qualificação;
- ***firstQualiPos***: Junção Canalizada na *M-Tree* com Primeira Qualificação e Posição da Rota;
- ***minQuali***: Junção Canalizada na *M-Tree* com Qualificação Mínima;
- ***minQualiPos***: Junção Canalizada na *M-Tree* com Qualificação Mínima e Posição da Rota;
- ***rTreeQuali***: Junção Canalizada na *R-Tree* com Primeira Qualificação e Posição da Rota;

É importante ressaltar que, de todos os algoritmos, a implementação sequencial foi utilizada apenas no último experimento (que utiliza de pequenas quantidades de dados), pois a sua execução é bastante lenta em relação às outras estratégias.

4.1 Experimento 1 (UNIFORME)

O Experimento 1 (UNIFORME) é o que possui maior quantidade de dados a serem processados pelos algoritmos de junção canalizada. Para este experimento, inicialmente foi criada uma matriz de 100.000.000 de pontos sintéticos que formam um conjunto de interesse (10.000 linhas $*$ 10.000 colunas).

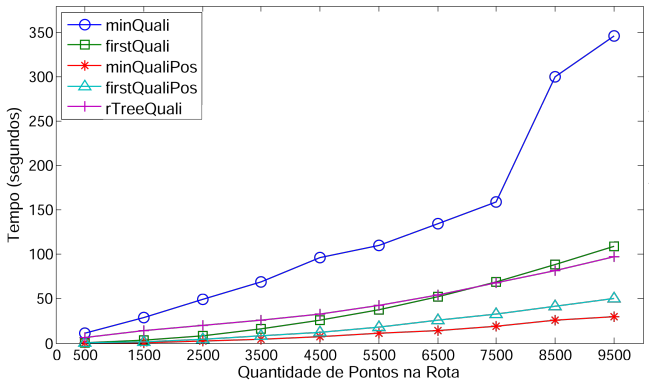
Foi gerado de forma sintética 9500 conjuntos de rotas, divididos em 10 grupos com 950 rotas cada. Para cada grupo, variou-se o quantitativo de pontos que formam a rota. O primeiro grupo possuía 500 pontos, o segundo 1500 pontos, e os outros grupos seguiram um incremento de 1000, até que o décimo grupo possuísse 9500 pontos.

Para geração destas rotas, um ponto da extremidade da matriz (conjunto de interesse) era selecionado aleatoriamente para se tornar o ponto inicial de uma rota. Em seguida, também de forma aleatória, este ponto seguia variando continuamente para uma mesma direção (cima, baixo, esquerda ou direita), até que a quantidade de pontos da rota fosse atingida.

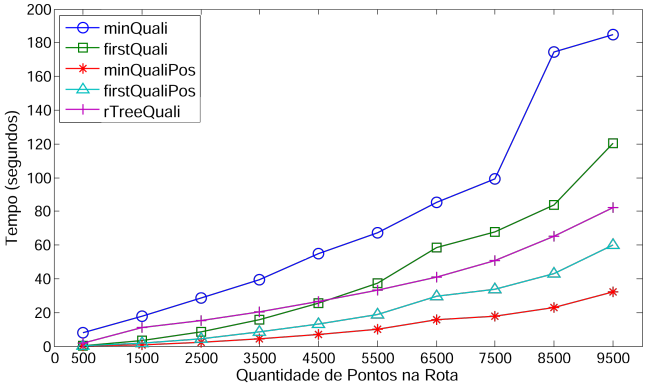
Por fim, para cada um dos grupos, extraiu-se a média das execuções da junção canalizada com largura de canal (ε) constante igual a 1 unidade.

A Figura 4.1 apresenta 6 gráficos dispostos em duas colunas e três linhas que descrevem os resultados deste experimento. As colunas representam os diferentes tamanhos de blocos (1KB e 2KB)

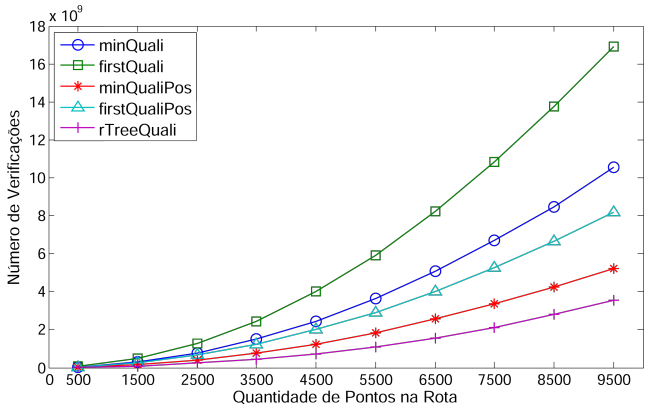
utilizados para execução dos experimentos. As linhas representam as médias de seus tempos de execução, número de verificações nas estruturas de dados e quantidade de acessos a disco, respectivamente.



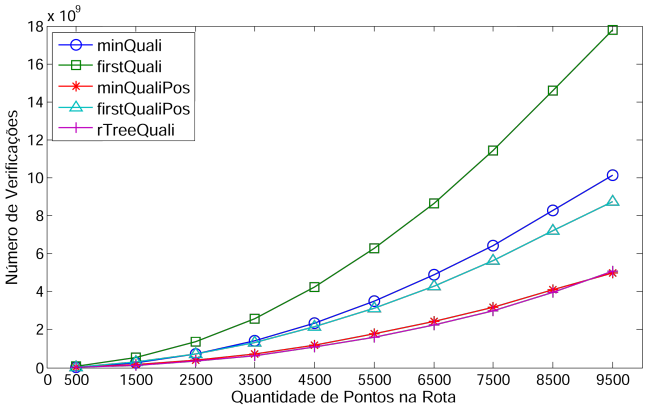
(a) Bloco 1KB - Tempos de execução.



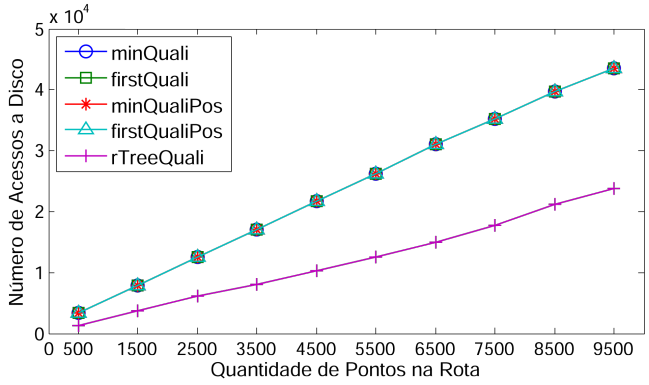
(b) Bloco 2KB - Tempos de execução.



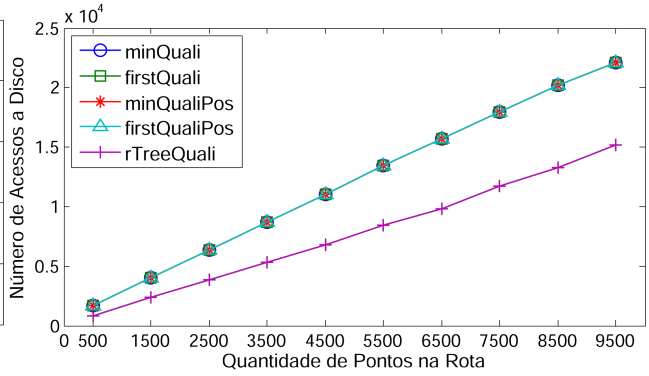
(c) Bloco 1KB - Verificações nas estruturas.



(d) Bloco 2KB - Verificações nas estruturas.



(e) Bloco 1KB - Quantidades de acesso a disco.



(f) Bloco 2KB - Quantidades de acesso a disco.

Figura 4.1: Experimento 1 (Uniforme) - Média dos tempos de execução, número de verificações nas estruturas de dados e quantidade de acessos a disco de cada um dos algoritmos.

Neste experimento, o algoritmo **minQualiPos** se destacou como o que possui o melhor tempo

de execução. Isso se deve ao fato deste algoritmo ser capaz de juntar dois recursos importantes apresentados neste trabalho: utilizar da posição da rota para evitar laços desnecessários ao avaliar um novo bloco e ser capaz de "podar" blocos com antecedência antes mesmo de serem avaliados mais em detalhes. Em seguida o algoritmo **firstQualiPos** se apresentou como o mais rápido, pois é capaz de interromper imediatamente o laço de rotas uma vez que um bloco é qualificado e também por utilizar da posição da rota para evitar laços desnecessários.

Em seguida, o algoritmo **firstQuali** foi melhor até o quantitativo de pontos na rota atingir 6500 para blocos de 1KB, e 5500 para blocos de 2KB, mas a partir destes quantitativos, foi superado pelo algoritmo **rTreeQuali**.

Neste experimento, os algoritmos **firstQuali** e **minQuali** obtiveram resultados inferiores no geral, pois utilizam de estratégias aplicadas de forma individual, neste caso, interromper o laço da rota ao qualificar um bloco e "podar" blocos com antecedência, respectivamente.

Os números de verificações neste trabalho remetem à quantidade de cálculos de distância que foram necessários para se calcular a distância entre os pontos pertencentes à rota e blocos responsáveis por armazenarem os pontos do conjunto de interesse. Nos algoritmos baseados na *M-Tree*, computou-se como verificação os cálculos de distância. Já na *R-Tree*, computou-se como verificação a contagem de sobreposições de MBR's (nos blocos índices) e cálculos de distância (realizados nos blocos folhas). Neste quesito, o algoritmo **rTreeQuali** realizou menos verificações que os demais algoritmos. Porém, observou-se que, com o aumento do tamanho dos blocos, o algoritmo **minQualiPos** tende a alcançar o algoritmo **rTreeQuali** em números de verificações.

No quesito número médio de acessos a disco, como já era esperado, conforme os blocos aumentam de tamanho, menos acessos a disco precisam ser realizados. No geral, o algoritmo **rTreeQuali** obteve melhor resultado que as demais implementações. Os algoritmos baseados na *M-Tree* tiveram o mesmo número de acessos a disco, pois como as estruturas de dados que indexam os pontos do conjunto de interesse são lidas no laço externo, não existe otimização (interrupção). As otimizações (interrupções) estão no laço interno, responsável por caminhar sobre os pontos da rota, que não faz acesso a disco, pois os pontos já se encontram inteiramente em memória principal.

4.2 Experimento 2 (UK Postcodes)

Este experimento é o segundo que possui maior quantidade de dados a serem processados pelos algoritmos de junção canalizada. O conjunto de interesse foi construído sobre uma base de dados real disponibilizada pela (DATA.GOV.UK, 2016), que contém latitudes e longitudes de todos os códigos postais do Reino Unido, totalizando 1.666.774 pontos distintos.

Foi gerado de forma sintética 9500 conjuntos de rotas, dividido em 10 grupos com 950 rotas cada,

da mesma forma que aconteceu no Experimento 1 (UNIFORME).

A sistemática de geração das rotas neste experimento funcionou da seguinte maneira: inicialmente era selecionada de forma aleatória uma posição geográfica da base de códigos postais do Reino Unido. Em seguida, era selecionada a extremidade de uma direção, dentre as 50 posições geográficas mais próximas da posição inicialmente selecionada. A extremidade desta direção sempre era a mesma em uma rota, ou seja, seguindo para cima (maior longitude), baixo (menor longitude), esquerda (menor latitude) ou direita (maior latitude). Essa sistemática de seleção de posições geográficas continuava até que a quantidade de pontos de uma determinada rota fosse atingida.

Por fim, para cada um dos grupos, extraiu-se a média das execuções da junção canalizada com largura de canal (ε) constante e igual a 100 metros;

A Figura 4.2 apresenta nas colunas os diferentes tamanhos de blocos (1KB e 2KB), e nas linhas, a média de seus tempos de execução, número de verificações nas estruturas de dados e quantidade de acessos a disco, respectivamente.

Neste experimento, o algoritmo **minQualiPos** mais uma vez se destacou como o que possui o melhor tempo de execução, porém, dessa vez juntamente do algoritmo **minQuali**. Como foi descrito anteriormente, ambos os algoritmos utilizam da estratégia de “poda” por desigualdade triangular, que neste experimento em especial, pode ter auxiliado no ótimo desempenho alcançado.

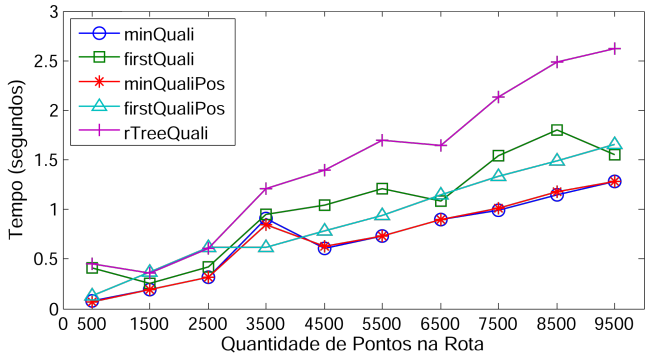
Os algoritmos **fistQuali** e **fistQualiPos** apesar de apresentarem resultados inferiores, em alguns momentos chegaram a atingir tempos similares ou muito próximos, como aconteceu no processamento de 6500 pontos da rota em ambos os tamanhos de bloco. Isso se deve ao fato de também utilizarem de uma estratégia em comum, no caso, de serem capazes de interromper o laço da rota ao qualificarem um bloco.

O algoritmo **rTreeQuali** se destacou como o mais lento, diferente do que aconteceu no Experimento 1, que chegou a atingir um resultado mediano. Acredita-se que este algoritmo poderia atingir melhores resultados caso também permitisse "podar" blocos índices com antecedência, assim como acontece nos algoritmos **minQuali** e **minQualiPos**. Porém, até o momento não foi encontrada uma técnica ou heurística que permita fazer isso.

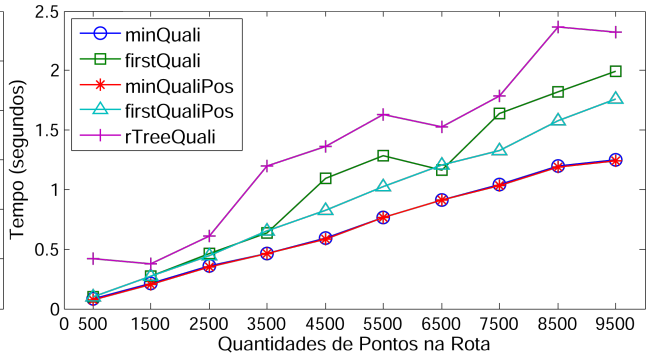
No quesito número de verificações, mais uma vez o algoritmo **rTreeQuali** se destacou como o que realizou o menor número. Porém, para os algoritmos baseados na *M-Tree*, seus respectivos números de verificações tenderam a acompanhar ou ficar próximo de suas posições obtidas no quesito tempo de execução.

No quesito número médio de acessos a disco, mais uma vez o algoritmo **rTreeQuali** obteve melhor resultado que as demais implementações. Os algoritmos baseados na *M-Tree* também tiveram o mesmo número de acessos a disco, pelo fato já explicado no Experimento 1 (UNIFORME), ou seja,

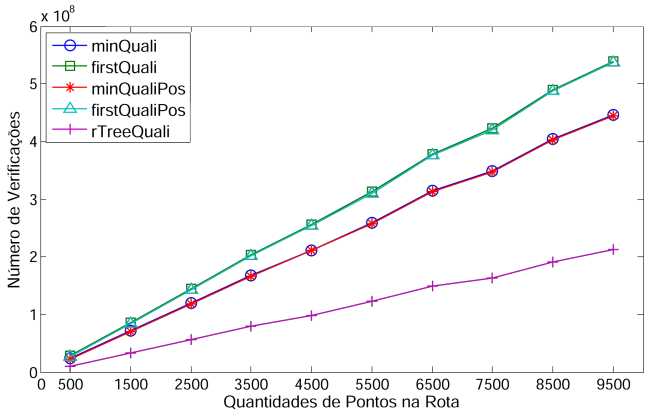
de não existir otimizações ou interrupções no laço mais externo.



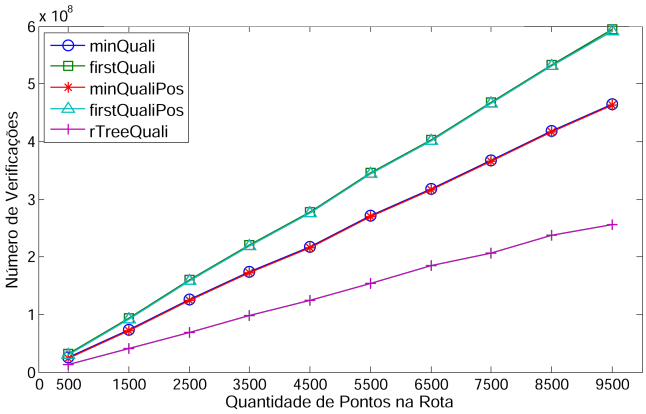
(a) Bloco 1KB - Tempos de execução.



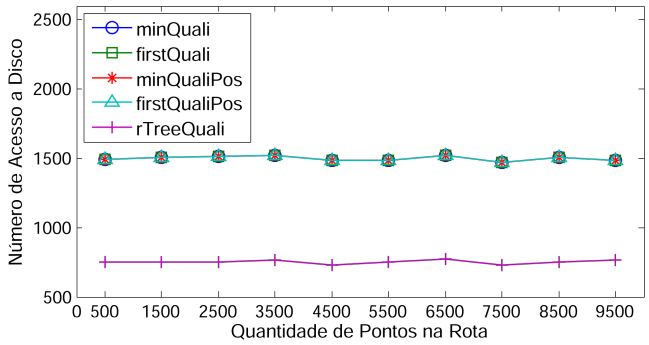
(b) Bloco 2KB - Tempos de execução.



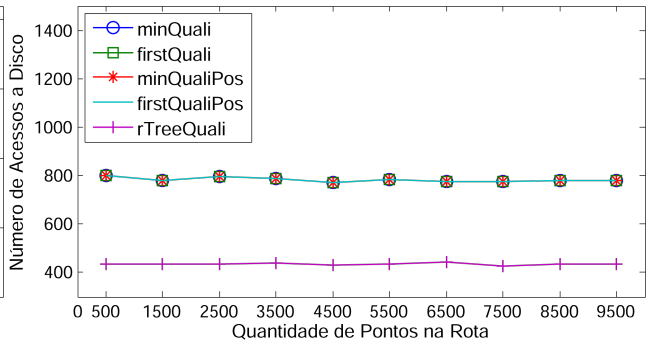
(c) Bloco 1KB - Verificações nas estruturas.



(d) Bloco 2KB - Verificações nas estruturas.



(e) Bloco 1KB - Quantidades de acesso a disco.



(f) Bloco 2KB - Quantidades de acesso a disco.

Figura 4.2: Experimento 2 (UK *Postcodes*) - Média dos tempos de execução, número de verificações nas estruturas de dados e quantidade de acessos a disco de cada um dos algoritmos.

4.3 Experimento 3 (ALOI)

Em relação aos Experimentos 1 (UNIFORME) e 2 (UK Postcodes), este experimento foi executado utilizando uma quantidade inferior de dados. Foi utilizado como conjunto de interesse imagens

pertencentes a uma base denominada ALOI (*Amsterdam Library of Object Images*) (IMAGES, 2004), formada por 110.250 imagens, que nada mais são que 110 variações de 1000 imagens originais. Essas 110 variações formam novas imagens com modificações nos ângulos de visão, iluminação e coloração de uma imagem original. Para cada imagem utilizada para formação do conjunto de interesse e do conjunto de rotas, foi utilizada a técnica denominada *Haralick features* (SETS, 2016) para extração de 13 características.

Diferente do que foi feito nos outros experimentos, neste optou-se por variar a quantidade de rotas e manter uma quantidade de pontos igual em cada uma delas. Foi selecionado 500 conjuntos de rotas, e cada uma delas possuía 56 vetores com 13 características de uma imagem. Cada um dos 56 vetores de características que formam uma rota pertence a uma variação de uma mesma imagem original, escolhida aleatoriamente.

Foram organizados 10 grupos de execução de junções. O primeiro grupo continha 5 rotas,o segundo 15, e assim foi sendo incrementando de 10 em 10, até que o décimo grupo atingisse 95 rotas. Também diferente do que foi feito nos outros experimentos, neste foi utilizado uma largura de canal (ε) diferente para cada rota. O valor de ε para cada rota foi convencionado utilizando da distância entre a primeira e a terceira imagem do conjunto de 110 variações de uma imagem original.

A Figura 4.3 também apresenta nas colunas os diferentes tamanhos de blocos (1KB e 2KB) e nas linhas, a média de seus tempos de execução, número de verificações nas estruturas de dados e quantidade de acessos a disco, respectivamente.

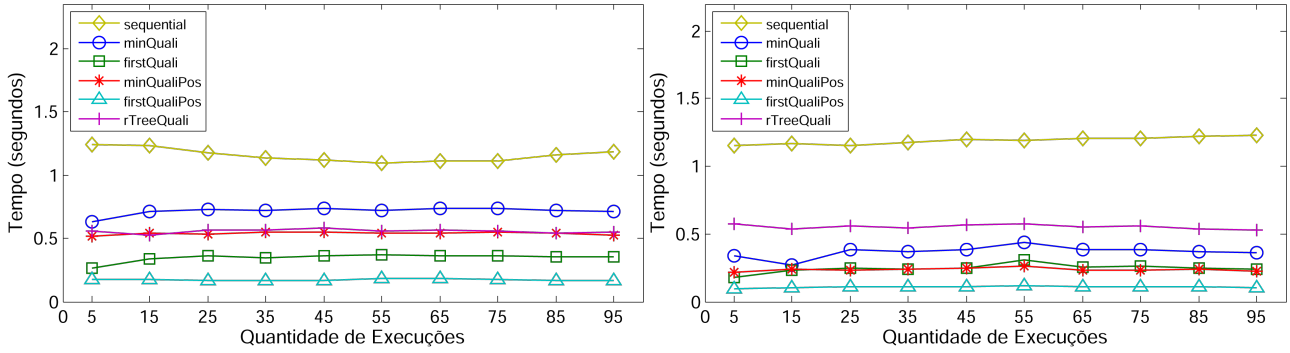
Em relação aos tempos médios de execução neste experimento, os algoritmos **firstQualiPos** e **firstQuali**, que antes apresentaram resultados inferiores ou medianos, agora atingiram o 1^o e 2^o melhor tempo, respectivamente. Isso mostra que, para experimentos muito pequenos, a interrupção imediata do laço de rotas que acontece quando um bloco é qualificado (presente nestes dois algoritmos), pode chegar a apresentar resultados similares ou superiores a outras estratégias utilizadas.

Apesar dos algoritmos **minQuali** e **minQualiPos** terem atingido tempos de execução inferiores quando executados sobre blocos de 1KB, observou-se que com o aumento do tamanho dos blocos, os tempos de execução destes algoritmos chegam a se aproximar dos tempos atingidos pelos algoritmos **firstQualiPos** e **firstQuali** neste experimento.

O tempo médio de execução do algoritmo **rTreeQuali** foi praticamente o mesmo, independente do tamanho dos blocos. Já o algoritmo sequencial, como já era de se esperar, obteve o pior tempo médio de execução. Isso acontece por conta deste algoritmo não utilizar de nenhuma estratégia específica para evitar processamentos desnecessários.

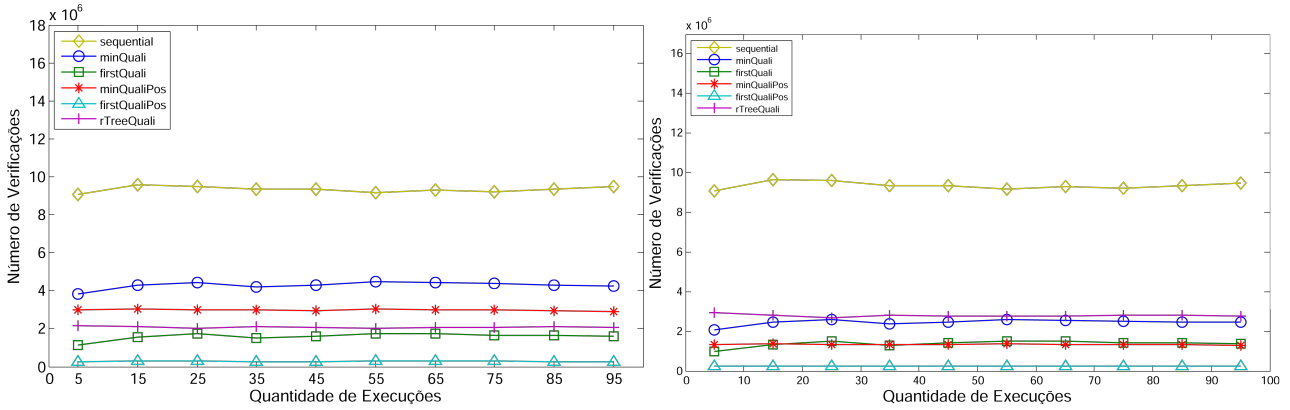
No quesito número de verificações, desta vez o algoritmo **rTreeQuali** não se destacou como o que realizou o menor número. Para todos os algoritmos, no geral, seus respectivos números de verificações

tenderam a acompanhar ou ficar próximo de suas posições obtidas no quesito tempo de execução.



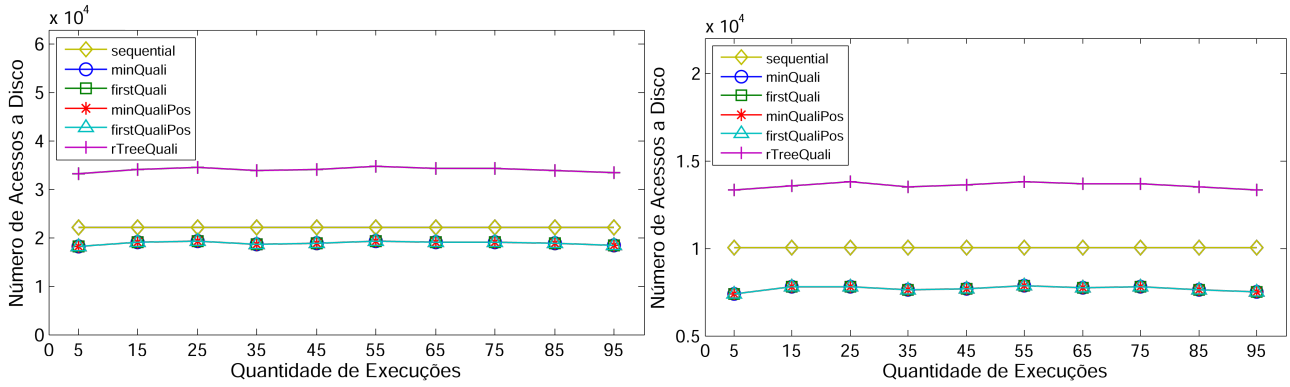
(a) Bloco 1KB - Tempos de execução.

(b) Bloco 2KB - Tempos de execução.



(c) Bloco 1KB - Verificações nas estruturas.

(d) Bloco 2KB - Verificações nas estruturas.



(e) Bloco 1KB - Quantidades de acesso a disco.

(f) Bloco 2KB - Quantidades de acesso a disco.

Figura 4.3: Experimento 3 (ALOI) - Média dos tempos de execução, número de verificações nas estruturas de dados e quantidade de acessos a disco de cada um dos algoritmos.

Por fim, no quesito número médio de acessos a disco, os algoritmos baseados na *M-Tree*, no geral, obtiveram melhores resultados quando aplicados em pequenos conjuntos de dados. Já o algoritmo **rTreeQuali**, que em outros experimentos foi superior, para pequenos conjuntos de dados chegou a

ser inferior até mesmo ao algoritmo *sequencial* no quesito número médio de acessos a disco.

Capítulo 5 Conclusão

Com o advento e popularização dos dispositivos móveis, cada vez mais dados estão sendo gerados pela sociedade. Por conta da alta conectividade destes dispositivos, grande parte destes dados acabam sendo armazenados em grandes bancos de dados, que precisam prover recursos cada vez mais refinados a seus usuários a fim de prover consultas cada vez mais rápidas e precisas. Para isso, os operadores disponíveis para consulta e manipulação dos bancos de dados precisam estar sempre evoluindo.

Este trabalho buscou explorar mais em detalhes um operador de junção denominado Junção Canalizada. Com este operador, pontos de interesse que se encontram em um espaço euclidiano e próximos a uma rota podem ser facilmente buscados, pois este operador é capaz de formar um canal virtual de largura ε capaz de qualificá-los, retornando assim, um novo conjunto resposta ao usuário.

Apesar deste operador ter sido proposto e validado por Duarte (2012), ele ainda não tinha sido formalizado por meio de uma linguagem de consulta formal para bancos de dados relacionais e nenhuma pesquisa até o momento tinha sido feita buscando aperfeiçoá-lo. Por isso, se tornou alvo de pesquisa deste trabalho.

Como principais contribuições deste trabalho, pode-se citar a formalização do operador de Junção Canalizada por meio do cálculo relacional de domínio, a implementação deste operador fazendo uso da estrutura de dados métrica *M-Tree* juntamente com a apresentação de 4 novas estratégias de execução sobre esta estrutura, e a implementação deste operador fazendo uso da estrutura de dados *R-Tree*.

Como contribuições secundárias deste trabalho pode-se citar a utilização do *framework Object-Injection* para implementação das novas estratégias propostas, a criação de novos pacotes neste *framework* que permitem a execução dos experimentos propostos de forma customizada e também a utilização de diferentes tipos de dados para formação dos pontos de interesse e rotas.

Por fim, como propostas de trabalhos futuros, sugere-se a avaliação das estratégias propostas em

outros espaços que não seja o euclidiano, a realização de aprimoramentos do algoritmo *rTreeQuali* e também a paginação dos resultados encontrados pela Junção Canalizada, a fim de retornar resultados aos usuários antes mesmo da execução da junção ser totalmente finalizada.

Referências Bibliográficas

BöHM, C. et al. Epsilon grid order: An algorithm for the similarity join on massive high-dimensional data. *SIGMOD Rec.*, ACM, New York, NY, USA, v. 30, n. 2, p. 379–388, maio 2001. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/376284.375714>.

BRINKHOFF, T.; KRIEGEL, H.-P.; SEEGER, B. Efficient processing of spatial joins using r-trees. *SIGMOD Rec.*, ACM, New York, NY, USA, v. 22, n. 2, p. 237–246, jun. 1993. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/170036.170075>.

CARVALHO, L. O. *Obinject-Injection: Um Framework de Indexação e Persistência*. Dissertação (Mestrado) — UNIFEI - Universidade Federal de Itajubá, March 2013.

CARVALHO, L. O. et al. Obinject: a noodmg persistence and indexing framework for object injection. *JIDM*, v. 4, n. 3, p. 220–235, 2013. Disponível em: <http://seer.lcc.ufmg.br/index.php/jidm/article/view/240>.

CIACCIA, P.; PATELLA, M.; ZEZULA, P. M-tree: An efficient access method for similarity search in metric spaces. In: *Proceedings of the 23rd International Conference on Very Large Data Bases*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 1997. (VLDB '97), p. 426–435. ISBN 1-55860-470-7. Disponível em: <http://dl.acm.org/citation.cfm?id=645923.671005>.

DATA.GOV.UK. *UK Postcodes*. 2016. <https://data.gov.uk/apps/uk-postcodes>. [Online; accessed 11-May-2016].

DENG, K. et al. Trajectory indexing and retrieval. In: *Computing with Spatial Trajectories*. Springer New York, 2011. p. 35–60. Disponível em: <https://doi.org/10.1007\%2F978-1-4614-1629-6_2>.

DITTRICH, J.-P.; SEEGER, B. Gess: A scalable similarity-join algorithm for mining large data sets in high dimensional spaces. In: *Proceedings of the Seventh ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. New York, NY, USA: ACM, 2001. (KDD '01), p. 47–56. ISBN 1-58113-391-X. Disponível em: <http://doi.acm.org/10.1145/502512.502524>.

DOHNAL, V.; GENNARO, C.; ZEZULA, P. Similarity join in metric spaces using eD-index. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2003. p. 484–493. Disponível em: <https://doi.org/10.1007\%2F978-3-540-45227-0_48>.

DUARTE, A. L. *Junção Canalizada*. Dissertação (Mestrado) — UNIFEI - Universidade Federal de Itajubá, January 2012.

ESMASRI, R.; NAVATHE, S. B. *Sistemas de Banco de Dados*. 6ª. ed. [S.l.]: PEARSON, 2005.

FERRO, C. *ObInject Query Language*. Dissertação (Mestrado) — UNIFEI - Universidade Federal de Itajubá, November 2012.

GUTTMAN, A. R-trees: A dynamic index structure for spatial searching. *SIGMOD Rec.*, ACM, New York, NY, USA, v. 14, n. 2, p. 47–57, jun. 1984. ISSN 0163-5808. Disponível em: <http://doi.acm.org/10.1145/971697.602266>.

IMAGES, A. L. of O. *ALOI*. 2004. <http://aloi.science.uva.nl/>. [Online; accessed 14-May-2016].

JACOX, E. H.; SAMET, H. Metric space similarity joins. *ACM Trans. Database Syst.*, ACM, New York, NY, USA, v. 33, n. 2, p. 7:1–7:38, jun. 2008. ISSN 0362-5915. Disponível em: <http://doi.acm.org/10.1145/1366102.1366104>.

KALASHNIKOV, D. V. Super-EGO: fast multi-dimensional similarity join. *The VLDB Journal*, Springer Nature, v. 22, n. 4, p. 561–585, feb 2013. Disponível em: <https://doi.org/10.1007\%2Fs00778-012-0305-7>.

KALASHNIKOV, D. V.; PRABHAKAR, S. Similarity join for low- and high- dimensional data. In: *In Proc. 8th Intl. Conf. on Database Systems for Advanced Applications (DASFAA'03*. [S.l.: s.n.], 2003. p. 7–16.

KALASHNIKOV, D. V.; PRABHAKAR, S. Fast similarity join for multi-dimensional data. *Inf. Syst.*, Elsevier Science Ltd., Oxford, UK, UK, v. 32, n. 1, p. 160–177, mar. 2007. ISSN 0306-4379. Disponível em: <http://dx.doi.org/10.1016/j.is.2005.07.002>.

KALASHNIKOV, D. V.; PRABHAKAR, S.; HAMBRUSCH, S. E. Efficient evaluation of continuous range queries on moving objects. In: *In DEXA 2002, Proc. of the 13th International Conference and Workshop on Database and Expert Systems Applications, Aix en Provence*. [S.l.: s.n.], 2002. p. 731–740.

KIM, W. (Ed.). *Modern Database Systems: The Object Model, Interoperability, and Beyond*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 1995. ISBN 0-201-59098-0.

LIAN, X.; CHEN, L. Set similarity join on probabilistic data. *Proc. VLDB Endow.*, VLDB Endowment, v. 3, n. 1-2, p. 650–659, set. 2010. ISSN 2150-8097. Disponível em: <http://dx.doi.org/10.14778/1920841.1920924>.

MANOLOPOULOS, Y. et al. *R-Trees: Theory and Applications*. [S.l.]: Springer Publishing Company, Incorporated, 2005. ISBN 1852339772, 9781852339777.

MISHRA, P.; EICH, M. H. Join processing in relational databases. *ACM Comput. Surv.*, ACM, New York, NY, USA, v. 24, n. 1, p. 63–113, mar. 1992. ISSN 0360-0300. Disponível em: <http://doi.acm.org/10.1145/128762.128764>.

PAREDES, R.; REYES, N. List of twin clusters: a data structure for similarity joins in metric spaces. In: *2008 IEEE 24th International Conference on Data Engineering Workshop*. IEEE, 2008. Disponível em: <https://doi.org/10.1109\%2Ficdew.2008.4498353>.

SERAPHIM, E. *Operadores Binários para consulta de similaridade em banco de dados multimídia*. Tese (Doutorado) — USP - São Carlos, Outubro 2005.

SETS, M.-V. D. *First 13 Haralick features*. 2016. <https://elki-project.github.io/datasets/multi_view>. [Online; accessed 14-May-2016].

SHEKHAR, S.; YOO, J. S. Processing in-route nearest neighbor queries. In: *Proceedings of the eleventh ACM international symposium on Advances in geographic information systems - GIS 2003*. ACM Press, 2003. Disponível em: <https://doi.org/10.1145\%2F956676.956678>.

SILVA, Y. N.; AREF, W. G.; ALI, M. H. The similarity join database operator. In: *Proceedings of the 26th International Conference on Data Engineering, ICDE 2010, March 1-6, 2010, Long Beach, California, USA*. [s.n.], 2010. p. 892–903. Disponível em: <http://dx.doi.org/10.1109/ICDE.2010.5447873>.

SKOPAL, T. et al. Revisiting m-tree building principles. In: *Advances in Databases and Information Systems, 7th East European Conference, ADBIS 2003, Dresden, Germany, September 3-6, 2003, Proceedings*. [s.n.], 2003. p. 148–162. Disponível em: <http://dx.doi.org/10.1007/978-3-540-39403-7_13>.

SONG, Z.; ROUSSOPOULOS, N. K-nearest neighbor search for moving query point. In: *Proceedings of the 7th International Symposium on Advances in Spatial and Temporal Databases*. London, UK, UK: Springer-Verlag, 2001. (SSTD '01), p. 79–96. ISBN 3-540-42301-X. Disponível em: <http://dl.acm.org/citation.cfm?id=647227.719093>.

TRAINA, J. C. et al. Fast indexing and visualization of metric data sets using slim-trees. *IEEE Transactions on Knowledge and Data Engineering*, IEEE Computer Society, Los Alamitos, CA, USA, v. 14, n. 2, p. 244–260, 2002. ISSN 1041-4347.

WANG, H.; ZIMMERMANN, R.; KU, W.-S. Distributed continuous range query processing on moving objects. In: *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2006. p. 655–665. Disponível em: <https://doi.org/10.1007\%2F11827405_64>.

WOLFE, H. E. *Introduction to Non-Euclidean Geometry*. [S.l.]: University of Michigan, 1945.

XUAN, K. et al. Continuous range search query processing in mobile navigation. In: *2008 14th IEEE International Conference on Parallel and Distributed Systems*. Institute of Electrical and Electronics Engineers (IEEE), 2008. Disponível em: <https://doi.org/10.1109\%2Ficpads.2008.69>.

YUAN, W.; SCHNEIDER, M. Supporting continuous range queries in indoor space. In: *2010 Eleventh International Conference on Mobile Data Management*. Institute of Electrical and Electronics Engineers (IEEE), 2010. Disponível em: <https://doi.org/10.1109\%2Fmdm.2010.21>.

ZEZULA, V. D. P.; AMATO, G. *Similarity Search The Metric Space Approach*. Kluwer Academic Publishers, 2006. Disponível em: <https://doi.org/10.1007\%2F0-387-29151-2>.

ZHOU, X.; CHEN, L. Event detection over twitter social media streams. *The VLDB Journal*, Springer Nature, v. 23, n. 3, p. 381–400, jul 2013. Disponível em: <https://doi.org/10.1007\%2Fs00778-013-0320-3>.

ZHUANG, Y. et al. Efficient batch similarity join processing of social images based on arbitrary features. *World Wide Web*, Springer Nature, v. 19, n. 4, p. 725–753, jul 2015. Disponível em: <https://doi.org/10.1007\%2Fs11280-015-0355-z>.