UNIVERSIDADE FEDERAL DE ITAJUBÁ

PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

OBJECT-INJECTION: UM FRAMEWORK DE INDEXAÇÃO E PERSISTÊNCIA

Luiz Olmes Carvalho

UNIFEI
Itajubá
Março, 2013

UNIVERSIDADE FEDERAL DE ITAJUBÁ
PROGRAMA DE PÓS-GRADUAÇÃO EM
CIÊNCIA E TECNOLOGIA DA COMPUTAÇÃO

Luiz Olmes Carvalho

OBJECT-INJECTION: UM FRAMEWORK DE INDEXAÇÃO E PERSISTÊNCIA

Dissertação de Mestrado 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.

Área de Concentração: Sistemas de Computação

Orientador: Prof. Dr. Enzo Seraphim

UNIFEI
Itajubá
Março, 2013

© 2013
Luiz Olmes Carvalho

Dados Internacionais de Catalogação na Publicação (CIP)

C331o Carvalho, Luiz Olmes.
Object-Injection: Um Framework de Indexação e Persistência / Luiz Olmes Carvalho. – Itajubá: UNIFEI, 2013.
xxv + 88 f. : il. ; 29.7

Orientador: Enzo Seraphim.
Dissertação (Mestrado em Ciência e Tecnologia da Computação) – Universidade Federal de Itajubá, Itajubá, 2013.
1. Persistência. 2. Indexação. 3. Framework. 4. NoODMG. I. Seraphim, Enzo, orient. II. Universidade Federal de Itajubá. III. Título.

CDU 004.65

Exemplar correspondente à versão final do texto, após a realização da defesa, incluindo, portanto, as devidas considerações dos examinadores.

Typeset in pdfLaTeX

Folha de Aprovação

Dissertação **aprovada** pela Banca Examinadora em 1º de março de 2013, conferindo ao autor o título de **Mestre em Ciência e Tecnologia da Computação**.

Dr. Enzo Seraphim ______________________________

Orientador

Universidade Federal de Itajubá

Dr^{a}. Isabela Neves Drummond ______________________________

Membro da Banca

Universidade Federal de Itajubá

Dr. Marcos Rodrigues Vieira ______________________________

Convidado

IBM Research Lab., Rio de Janeiro, Brazil

Mas as coisas findas
muito mais que lindas,
essas ficarão.

Carlos Drummond de Andrade

A meus queridos pais

Agradecimentos

A seção de agradecimentos é onde corremos o risco de cometer alguma injustiça do tipo esquecer de alguém. É uma situação embaraçosa, pois o esforço para produzir este trabalho foi expressivo e, portanto, todo e qualquer auxílio foi bem recebido. Ainda assim, não quero aqui deixar de registrar os nomes das boas almas que, de alguma maneira, ajudaram-me durante o curso de pós-graduação.

Ao que imprescindível, incomparável e inconfundível na *Sua* infinita bondade, refúgio e porto seguro nos momentos de tempestades e ventos contrários, *Àquele* que mantém o barco da vida em curso retilíneo, não permitindo que se desgarre para a imensidão do infinito ou naufrague em águas turvas, *Àquele* que compreende os meus anseios e dá a necessária coragem para atingir meus objetivos, dirijo com o coração grato, ofereço o meu porvir e peço sabedoria, a *Sua* sabedoria, para sempre agir com eficiência no trabalho e acerto nas decisões.

A meus pais, *Ivo Guimarães de Carvalho* e *Maria Lades da Silva Carvalho*. A vocês, que deram-me a vida e ensinaram-me a vivê-la com dignidade. A vocês, que iluminaram os caminhos obscuros com afeto e dedicação para que os trilhasse sem medo e cheio de esperanças. A vocês, que se doaram inteiros e renunciaram aos seus sonhos, para que, muitas vezes, pudesse realizar os meus. Pela longa espera e compreensão durante esses longos anos. A vocês, pais por natureza, por opção e por amor, não bastaria dizer, que não há palavras para agradecer tudo isso. Mas é o que acontece agora, quando procuro arduamente uma forma verbal de exprimir uma emoção ímpar. Uma emoção que jamais seria traduzida por palavras. Amo vocês! Este mestrado é dedicado a vocês, os mestres de minha vida!

A meu professor, estimado amigo, anjo da guarda e orientador, *Prof. Dr. Enzo Seraphim*, a quem devo, além da orientação deste trabalho, o aprendizado e amadurecimento de diversos conceitos, teóricos e práticos, temas desta dissertação. Agradeço pela orientação precisa, segura e amiga; pela disponibilidade e empenho pessoal e profissional manifestados. Por me fazer crescer como estudante e como pessoa. Por me mostrar o valor da pesquisa científica e me ensinar a ser criterioso e meticuloso em meus trabalhos, sobretudo com os resultados. Pelo incentivo, atenção, apoio e encorajamento. Por não desistir e acreditar em mim em todos os momentos, principalmente nos mais críticos, onde eu, mesmo sem deixar transparecer, não mais acreditava. Pela sua extrema gentileza e lições de compreensão e humanismo, sem as quais, certamente eu iria para o "lado negro da força". Obrigado por me apontar o "caminho da luz"; pela sua admirável nobreza, que o faz enxergar apenas o bem em cada pessoa. Agradeço também pelos "vudus" de código e "coelhos tirados da cartola" (árvore R, reflexão, instância do parâmetro genérico...). O senhor é um exemplo de primor e otimismo, modelo de pesquisador a ser seguido pelos seus alunos. Nunca te disse isso pessoalmente, mas vários

colegas de classe já me ouviram falar que dou-me por satisfeito se um dia eu for metade do profissional que o senhor é. Perdoe meus momentos de teimosia e imaturidade, sei que o senhor só queria o meu bem. Espero, com toda sinceridade, que este seja o primeiro de muitos trabalhos que faremos juntos. Agradeço também à *Profª. Drª. Thatyana de Faria Piola Seraphim*, sempre amiga, atenciosa, gentil, receptiva e bondosa. Conversar com a senhora e contar "causos" é muito divertido! Ao pequeno *Miguel*, futuro computólogo(!), sempre calmo e paciente em meus braços desajeitados.

Ao *Prof. Dr. Edmilson Marmo Moreira*. Começo agradecendo pelo apoio dado no dia 20 de setembro de 2008, quando o Jonas Henrique, a Lais Vilela e eu, ainda na ingenuidade do 3º ano de graduação, fomos para a Maratona de Programação, em Santa Rita do Sapucaí. Ao fim daquele dia, numa noite chuvosa, em frente à portaria da Unifei, o senhor, o Jonas e eu dividíamos 2 guarda-chuvas e acabamos todos molhados. A água secou, se foi, mas as suas palavras de incentivo ficaram. Agradeço por nossas conversas abertas e transparentes, onde o senhor me colocava diante da realidade e me mostrava o dia de amanhã. O senhor me ensinou que primeiro as portas devem ser abertas, para depois tomarmos uma decisão. Ensinou-me a controlar a euforia e não agir dominado pela impulsividade. Aprendi, ainda, a não ter arrependimento; a não olhar para trás e não pensar em como as coisas teriam sido diferentes se outras escolhas fossem tomadas. Apesar de breve, nosso convívio foi marcante.

Ao *Prof. Dr. Laércio Augusto Baldochi Júnior*. Apesar de termos convivido apenas durante as disciplinas da graduação e da pós-graduação, reconheço-o como um profissional muito dedicado, sensato e correto. Agradeço também pelas cartas de recomendação.

Ao *Prof. Dr. Carlos Henrique Valério de Moraes ("Carlão")*, sempre carismático e motivador. Agradeço pela nossa conversa, quando eu procurava por um orientador, e o senhor me apontou o Enzo.

Ao *Prof. Dr. Otávio Augusto Salgado Carpinteiro*, por ser sempre gentil e confiar em mim. Agradeço também por permitir que eu fosse orientado do Enzo, mesmo quando a cota de alunos dele estava pra lá de estourada!

Ao *Prof. Dr. Carlos Alberto Murari Pinheiro*. Agradeço pelo interesse manifestado em minha pessoa e o reconhecimento de meu trabalho, ainda durante o período de disciplinas. Sinto não termos nos conhecido melhor e não ter ido trabalhar com o senhor, mas eu não fui feito para Inteligência Artificial.

A vocês, velhos amigos, sempre em ordem alfabética, como as coisas devem ser, *Helias de Menezes Pinto Soares*, *Henrique Édson Ramos Soares*, *Jonas Henrique Mendonça*, *Jussara Miranda Dias*, *Kleber Anderson Corrêa e Silva*, *Lais Reis Vilela* e *Luiz Felipe Gonçalves ("Paulo")*. Vocês são pessoas formidáveis. Fico feliz pelo privilégio de ter compartilhado de suas presenças e mais ainda por ver cada um trilhando um caminho de sucesso. Sinto falta de vocês por perto.

Como diz o poeta, bons amigos são a família que nos permitiram escolher. Considero-me iluminado por vocês serem tão numerosos em minha vida. A vocês, *Caio Henrique Lima Ribeiro*, *Elaine Cristina da Cunha*, *Hélio Baptista Martins*, *João Pedro Sarno Carvalho ("Grande")*, *Luiz Henrique de Paula Assis Pereira* e *Wellington dos Santos Flauzino ("isomórfico")*. Agradeço pela amizade, paciência, sabedoria, apoio, ensinamentos, conselhos, risos, confiança, momentos felizes, momentos não tão felizes, incentivo, aprendizado, enfim, por estarem presentes ao longo desta caminhada e, em meio à correria, ainda me dedicarem um tempo.

Aos colegas de classe, sem nomear ninguém para não cometer a injustiça do esquecimento. Nossos caminhos se cruzaram diante de um ideal comum. Partilhamos cada descoberta, desafio e conquista. Dividimos medos, incertezas e inseguranças... Mas somamos entusiasmo, forças e alegrias... Soubemos conviver e respeitar-nos ainda que nem sempre compartilhássemos as mesmas ideias. Lutamos, sobrevivemos, crescemos... Acima de tudo, como seres humanos. E por tudo, a saudade há de ficar.

A meus familiares, que sempre me desejam o melhor. Agradeço especialmente a meus tios *Arailton Alves da Silva*, *Carla Ângela da Silva*, *Ivone Veloso*, *José Agostinho Veloso ("tio Zé")*, *Marina de Carvalho Goulart*, *Nemir Fátima* e *Vilma Margarete Silva*. A meu avô e padrinho, *João Alves da Silva*. À *Maria de Lourdes dos Santos ("Léa")*, por todo o carinho.

Ao *Prof. Dr. Isaías de Lima*, *Profª. M. Sc. Katia Cristina Lage dos Santos*, *Prof. Dr. Enzo Seraphim* e à atual turma do *3º ano de Engenharia da Computação* (ECO 2011), pela oportunidade do estágio de docência.

Ao *Prof. Dr. Danilo Henrique Spadoti* e *Prof. M. Sc. Rodrigo Maximiano Antunes de Almeida*, pessoas finíssimas. Muito proveitosos os bons momentos em que convivemos!

À *Coordenação de Aperfeiçoamento de Pessoal de Nível Superior*, pelo auxílio financeiro. Aos professores da *Universidade Federal de Itajubá*, pelo conhecimento compartilhado e à universidade, pelo uso de suas instalações, recursos e equipamentos. A meus professores da graduação, que torcem por mim e sempre que nos encontramos, dedicam-me toda a atenção.

Aos funcionários do Instituto de Engenharia de Sistemas e Tecnologia da Informação e à *Regina Aparecida Salomon Storino*, do Departamento de Registro Acadêmico, sempre bem-humorados e dispostas a prestar auxílio.

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

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

Por fim, desejo aos colegas que trilham e aos que venham a trilhar este caminho, que encontrem o mesmo ambiente de amizade, atenção e apoio comigo compartilhado.

Resumo

O modelo orientado a objetos tem se tornado o padrão para desenvolvimento de sistemas de informação. Isto faz com que sejam criadas cada vez mais classes de negócio, que são instanciadas inúmeras vezes e geram uma superpopulação de objetos. Em muitos cenários, esses objetos devem mudar seu estado de transiente para tornar-se persistentes. Essa necessidade por persistência de objetos faz com que surjam diversas soluções não padronizadas pelo *Object Data Management Group* (ODMG). Algumas dessas soluções utilizam *frameworks* para realizar o mapeamento de objetos para relações em Sistemas Gerenciadores de Banco de Dados (SGBD). Tais *frameworks*, vinculados à camada de aplicação do usuário, delegam-na a responsabilidade de definir quais objetos ou classes usuárias serão persistidas. Entretanto, são raros os *frameworks* de persistência orientados a objetos que lidam com a criação de índices, usando estruturas como a Árvore B, Árvore M e Árvore R. Este trabalho apresenta um *framework* NoODMG para indexação e persistência de objetos. A persistência de objetos é realizada usando índices primários e a indexação de chaves através de índices secundários. A principal característica do *framework* proposto é permitir que os objetos sejam injetados em quaisquer estruturas de dados, que podem estar armazenadas em quaisquer dispositivos. Este *framework* está dividido em quatro módulos baseados em padrões de projeto. De acordo com os experimentos, suas abstrações alcançaram melhorias significativas de desempenho em relação à outras alternativas. Além disso, seu mecanismo de persistência não necessita de SGBDs Relacionais.

Palavras-chave: Persistência, Indexação, *Framework*, NoODMG.

Abstract

Object Injection: An Indexing and Persistence Framework

The object-oriented model has become the standard for developing information systems. Because of this, more and more business classes have been created and instantiated plenty of times, generating a super population of objects. In several scenery, these objects must change their state from transient to persistent. Thus, several solutions not standardized for the Object Data Management Group (ODMG) have been created to attend the need for objects persistence. Some of these solutions use frameworks to perform the mapping from objects to relations on the Database Management Systems (DBMS). Such frameworks, bound to user application tier, delegate to this tier the definition of which objects or user classes will be persisted. However, few objects persistence frameworks deal with indices creation, using structures like B-Tree, M-Tree and R-Tree. This work presents a NoODMG framework for objects indexing and persistence. Objects persistence is performed using primary indices and keys indexing are done by secondary indices. The main feature of this framework is to permit that objects being injected and indexed in any data structures which can be stored in any devices. This framework is divided into four modules based on Design Patterns. Their abstractions have achieved a significative performance improvement in relation to existent alternatives, according to experiments. In addition, its persistence mechanism does not need of Relational DBMSs (RDBMS).

Keywords: *Persistence, Indexing, Framework, NoODMG.*

Sumário

Lista de Figuras **xx**

Lista de Tabelas **xxi**

Abreviaturas e Siglas **xxiii**

Símbolos **xxv**

1 Introdução **1**

1.1 Motivação 3

1.2 Objetivos 4

1.3 Organização do trabalho 5

2 Métodos de Acesso **7**

2.1 Métodos de acesso espaciais 7

2.1.1 Árvore B 8

2.1.2 Hash Extensível 11

2.1.3 Árvore R 13

2.2 Métodos de acesso métricos 17

2.2.1 Árvore M 17

2.2.2 Árvore Slim 22

2.3 Considerações finais 25

3 Padrões em *Frameworks* **27**

3.1 *Frameworks* 27

3.2 Padrões de projeto 30

3.2.1 *Bridge* 31

3.2.2 *Decorator* 32

3.2.3 *Factory Method* 35
3.2.4 *Template Method* 37
3.2.5 *Curiously Recurring Template Pattern* 38
3.3 Considerações finais 40

4 O Framework Object-Injection **41**
4.1 Conceitos iniciais 41
4.1.1 Índices primários e secundários 42
4.1.2 Identificador Único Universal 42
4.2 Arquitetura modularizada 43
4.3 O módulo das Metaclasses 44
4.4 O módulo de Armazenamento 48
4.5 O módulo de Blocos 49
4.5.1 Nós da estrutura *Sequential* 52
4.5.2 Nós da estrutura *EHash* 53
4.5.3 Nós da *B-Tree* 54
4.5.4 Nós da *R-Tree* 54
4.5.5 Nós da *M-Tree* 55
4.6 O módulo de Dispositivos 55
4.7 Trabalhos similares 57

5 Experimentos **61**
5.1 Metodologia 61
5.2 Resultados 63

6 Conclusão **69**

Referências Bibliográficas **71**

A A Métrica de Xu **81**

Lista de Figuras

2.1 Representação de uma Árvore B . . . 8

2.2 Nó índice de uma Árvore B . . . 9

2.3 Representação de uma Árvore B+ . . . 10

2.4 Representação da Árvore B* . . . 11

2.5 Exemplo de *Hash* Extensível . . . 12

2.6 O diretório dobra após inserir 1010 na Figura 2.5. . . . 12

2.7 Exemplo de MBRs e objetos, onde os MBRs se interceptam mas os objetos não. 13

2.8 Representação da Árvore R . . . 14

2.9 Representação da Árvore R* . . . 16

2.10 Representação da Árvore M . . . 18

2.11 Consultas na Árvore M . . . 19

2.12 Regiões do espaço métrico nas Árvores M e PM . . . 22

2.13 Exemplo de *split* na Árvore Slim . . . 23

2.14 Aplicação do algoritmo *Slim-Down* . . . 24

2.15 Aplicação do algoritmo *Generalized Slim-Down* . . . 25

3.1 Exemplo do padrão *Bridge* . . . 32

3.2 Contexto de aplicação do padrão *Decorator* . . . 33

3.3 Extensão do modelo de classes da Figura 3.2 . . . 33

3.4 Exemplo do padrão *Decorator* . . . 34

3.5 Contexto de aplicação do padrão *Factory Method* . . . 36

3.6 Exemplo do padrão *Factory Method* . . . 37

3.7 Exemplo do padrão *Template Method* . . . 38

3.8 Modelagem do CRTP . . . 39

4.1 Organização dos módulos . . . 44

4.2 Hierarquia do módulo das Metaclasses 45
4.3 Hierarquia de serialização 47
4.4 Hierarquia do módulo de Armazenamento 49
4.5 Estrutura do Bloco 50
4.6 Modelagem do Node 50
4.7 Hierarquia do módulo de Blocos 51
4.8 Organização e customização dos blocos 53
4.9 Hierarquia do módulo de Dispositivos 55

5.1 Distribuição dos pontos da base da dados GEONet 62
5.2 Experimento: conjunto de dados do OpenOffice 64
5.3 Experimento: conjunto de dados UniProt 66
5.4 Experimento: conjunto de dados GEONet 67

A.1 Matriz da distância de Levenshtein 82
A.2 Cálculo da distância de Levenshtein entre *"tuesday"* e *"thursday"* 83

Lista de Tabelas

5.1 Tamanho dos arquivos (*bytes*) 68

A.1 Matriz mPAM 85

Abreviaturas e Siglas

AGM	–	Árvore Geradora Mínima
AOP	–	Aspect-Oriented Programming
CRTP	–	Curiously Recurring Template Pattern
DBMS	–	Database Management Systems
GUID	–	Globally Unique IDentifier
I/O	–	Input/Output - operação de Entrada/Saída
MAC	–	Media Access Control
MBR	–	Minimum Bounding Rectangle
mPAM	–	Matrix Point Accepted Mutation
ODMG	–	Object Data Management Group
OID	–	Object IDentifier
OMG	–	Object Management Group
OQL	–	Object Query Language
ORM	–	Object-Relational Mapping
RDBMS	–	Relational Database Management Systems
SGBD	–	Sistema Gerenciador de Banco de Dados
SGBDR	–	Sistema Gerenciador de Banco de Dados Relacional
SQL	–	Structured Query Language
URN	–	Uniform Resource Name
UTC	–	Universal Time Coordinated
UUID	–	Universally Unique IDentifier

Símbolos

d – função de distância
k_i – chave do objeto i
ne – número de entradas
o_i – objeto i
o_n – novo objeto
o_p – objeto representativo
p_i – referência do objeto i na base de dados
ptr_i – referência da sub-árvore da chave k_i
r_i – raio de cobertura do objeto i

Capítulo 1

Introdução

Concebido no início da década de 1960, nos laboratórios da *General Electric*, um *Sistema Gerenciador de Banco de Dados* (SGBD) é uma aplicação de *software* de uso geral que facilita o processo de definição, construção, manipulação e compartilhamento de dados entre diversos usuários e aplicações (ELMASRI; NAVATHE, 2011). Mais ainda, um SGBD fornece acesso a dados de forma estruturada e gerenciada, definindo também a *estrutura modelo* do dado armazenado.

Os primeiros SGBDs da década de 1960 estruturavam seus dados seguindo um *modelo de dados em rede*. O modelo de dados em rede permitia a um registro fazer referência a qualquer outro registro, formando uma rede (HELLAND, 2009). Depois de 20 anos, este modelo daria origem ao modelo de dados orientado a grafos, que, novamente, cairia em desuso, ressurgindo outra vez na década de 2000 (ANGLES; GUTIERREZ, 2008).

No final da década de 1960, juntamente com o *Information Management System*, um produto comercial da IBM na área de armazenamento de dados, surge o *modelo de dados hierárquico*. Neste modelo, os dados são orientados por uma hierarquia, e a forma de navegação entre os dados é dada em níveis, através do relacionamento entre o pai e o filho.

No início da década de 1970, com base no trabalho de Codd (1970) e num projeto de pesquisa da IBM conhecido como *System R* (ASTRAHAN et al., 1976), surge o *modelo relacional*. No modelo relacional, os dados são representados através de relações. Uma relação é definida como um conjunto de tuplas que possuem os mesmos atributos. Relações são representadas por tabelas, onde cada linha é uma tupla e cada coluna contém um atributo. Devido à disposição dos dados em tuplas, no modelo relacional não há ligações explícitas entre uma tupla e outra. Tuplas de diferentes tabelas podem ser relacionadas através de uma operação especial conhecida por *junção*.

O *System R*, por sua vez, deu origem à linguagem SQL, que tornou-se a linguagem padrão de manipulação de bancos de dados relacionais.

Já no início dos anos 1990, uma organização intitulada *Object Database Management Group* - que, mais tarde, se tornaria apenas OMG (*Object Management Group*) - propôs uma lista de especificações para os desenvolvedores escreverem suas aplicações. Como especificado nesta listagem, diferentes aplicações deveriam executar usando um esquema de dados e linguagens de consulta, manipulação e programação comuns. Esta lista de especificação deu origem ao *padrão ODMG* (CATTELL; BARRY, 2000). Em menos de uma década, aproveitando-se de contribuições (BARRY; STANIENDA, 1998) e discussões (CAREY; DEWITT, 1996; ALAGIC, 1997), o comitê ODMG expandiu seu foco e passou a cobrir os padrões de desenvolvimento de bancos de dados orientados a objetos e *frameworks* de mapeamento objeto-relacional. No início dos anos 2000, quando o padrão ODMG alcançou um nível maior de maturidade, a comunidade ODMG concentrou-se em difundir seu uso na indústria. O esforço da comunidade ODMG foi tamanho que fez, inclusive, com que fosse criada a especificação ODMG para armazenamento de objetos Java, conhecida como *Java Data Objects* (JDO) (JSR-12, 2012).

Apesar da inovação introduzida pelo padrão ODMG, ele não foi amplamente disseminado devido às diferenças entre organizações, aplicações, produtos legados anteriores ao padrão e, principalmente, por não tratar a indexação de dados. Assim, empresas, usuários médios e finais e desenvolvedores adotaram soluções de persistência que não seguiam, ou seguiam apenas parcialmente, o padrão ODMG.

Ainda nos anos 2000, os trabalhos de Chang et al. (2006) e DeCandia et al. (2007) impulsionaram o surgimento de algumas soluções de bancos de dados não-relacionais, conhecidas como *NoSQL (Not Only SQL)*. De acordo com Thomson e Abadi (2010), estas soluções relaxam as propriedades ACID a fim de garantir a escalabilidade no armazenamento de tipos de dados específicos. Originalmente, os dados gerenciados por soluções NoSQL eram baseados nos modelos de par chave-valor, tabular por colunas, orientado a grafos e orientado a documentos. Contudo, Leavitt (2010) sugere que a comunidade NoSQL está desorientada, tentando se expandir e englobar tudo que não é coberto por um padrão, como, por exemplo, os bancos de dados orientados a objetos.

Outras soluções que estão se tornando populares para o armazenamento de dados são os *frameworks* de persistência, como discutido na Seção 4.7. Nesse contexto, este trabalho apresenta um *framework* de indexação e persistência orientado a objetos, chamado *Object-Injection*. A principal característica deste *framework* é permitir que quaisquer objetos definidos em linguagem de programação sejam indexados por estruturas de dados que, por sua vez, podem estar armazenadas em diferentes dispositivos.

O *framework* proposto neste trabalho apresenta seu próprio esquema de definição e armazenamento de dados não se enquadrando, portanto, em nenhum dos modelos citados. Ainda que orientado a objetos, seguindo as ideias de Leavitt (2010), aplicações orientadas a objeto não pertencem originalmente ao padrão NoSQL, de modo que a classificação mais próxima e apropriada seria um *framework “NoODMG”*.

1.1 Motivação

Uma vez que, atualmente, há um grande número de bancos de dados relacionais disponíveis, muitos inclusive gratuitos, pode-se perguntar ou argumentar contra a necessidade de criação de mais um mecanismo de persistência. Os principais motivos que levaram ao desenvolvimento do *framework* Object-Injection foram as limitações dos Sistemas Gerenciadores de Banco de Dados Relacionais (SGBDRs) e a complexidade dos *frameworks* de persistência existentes.

Muitas das aplicações que armazenam seus dados em bancos utilizam bancos de dados relacionais, enquanto que suas classes de negócio são escritas em linguagem orientada a objetos. Como um banco relacional dispõe os dados em tabelas, que são inconsistentes com o modelo de classe das aplicações, os objetos de negócio devem ser mapeados para as relações predefinidas no banco. Esta necessidade em mapear objetos para tabelas e vice-versa consome tempo e resulta em uma perda de desempenho das aplicações ao usarem bancos relacionais.

A interação com bancos de dados relacionais requer uma certa familiaridade com uma linguagem especial, conhecida por SQL (*Structured Query Language*). O uso de SQL, contudo, introduz uma série de complicações.

A gramática da linguagem SQL é baseada no paradigma de programação declarativo, enquanto que as modernas linguagens de programação usam o paradigma imperativo. A programação declarativa descreve *o que* um programa deve fazer, mas sem descrever como suas ações e procedimentos funcionam. Por outro lado, o paradigma imperativo descreve as sequências, os comandos, o *como* realizar uma ação de forma específica (SEBESTA, 2001). Esta diferença de paradigmas acaba por impor mais incompatibilidade entre uma aplicação e um banco relacional e complicações para o desenvolvedor.

Outro problema da linguagem SQL é que, embora ela seja de certo modo padronizada, diferentes bancos relacionais trazem personalizações desta linguagem. Essas variações de sintaxe, ainda que mínimas, são suficientes para garantir que um código SQL de um banco não funcione como deveria quando portado para outro banco. Esta disparidade entre a linguagem SQL e as extensões proprietárias da SQL torna problemática a migração de aplicações completas de um banco para outro. Assim, soluções de persistência que usam SGBDRs devem mapear as diferenças de implementações da linguagem SQL.

Uma alternativa aos bancos de dados relacionais são os bancos de dados orientados a objetos. Diversas soluções existentes possuem uma semântica muito próxima à linguagem de programação orientada a objetos, tornando relativamente fácil armazenar, recuperar e gerenciar objetos ao invés de linhas de dados. Ainda que o emprego destas soluções tenha crescido, devido ao domínio das linguagens orientadas a objeto, elas também apresentam suas desvantagens. Embora exista um padrão de desenvolvimento de bancos de dados orientados a objetos

(CATTELL; BARRY, 2000), em termos práticos, muitos produtos implementam uma pequena parte do padrão. Além disso, soluções distintas apresentam amplas variações de quais partes do padrão elas implementam. Outro problema com os bancos orientados a objetos é que, para manipular seus dados, eles fazem uso de uma linguagem conhecida por OQL (*Object Query Language*), tão complexa quanto a SQL. Devido a esta complexidade, diferentes bancos implementam seu próprio conjunto de comandos OQL, levando ao mesmo problema de diversidade da SQL.

Quanto à complexidade dos *frameworks*, apesar da existência de muitos *frameworks* de persistência, como exemplificado na Seção 4.7, nenhum deles realiza a persistência de maneira transparente. Se um desenvolvedor decidir usar alguma implementação, por exemplo, da JPA (JSR-220, 2012), ele precisa fornecer uma enorme quantidade de informações, especificando o que e como ele quer persistir. Normalmente, documentações extensas ou livros inteiros necessitariam de serem lidos visando tirar o máximo proveito das vantagens oferecidas por estes *frameworks*. Além disso, muitos desses *frameworks* fazem uso do mapeamento objeto-relacional, apresentando as mesmas limitações recém discutidas.

1.2 Objetivos

O principal objetivo deste trabalho é desenvolver um *framework* de indexação e persistência orientado a objetos. Este *framework* visa ser facilmente acoplado a uma aplicação existente, oferecendo a persistência de forma nativa e dando ao programador liberdade para se concentrar em sua lógica de negócios, sem se preocupar com os detalhes de como os objetos são armazenados. Mais ainda, este *framework* fornece mecanismos base que auxiliam no desenvolvimento e implementação de novas estruturas de indexação de dados.

O *framework* proposto abrange as seguintes características:

- Fraco acoplamento de classes: as classes do usuário devem ser manipuladas diretamente pela solução de persistência. O acoplamento entre as classes do usuário e o *framework* é realizado sem que haja redundância de informação, visando poupar recursos computacionais.

- Flexibilidade de armazenamento: o armazenamento das entidades persistentes pode ser realizado relaxando ou não as propriedades ACID. O relaxamento das propriedades ACID garante um melhor desempenho para aplicações que não necessitam de processamento de transação. As formas de armazenamento podem ser abstrações de disco, memória, cliente-servidor, computação em grade, computação em nuvem etc.

- Identificação de objetos: objetos são marcados com um valor de identificador único, que permite a recuperação dos mesmos, independente do dispositivo de armazenamento.

- Indexação múltipla: estruturas de índice são usadas para responder à consultas do usuário de forma eficiente. Normalmente, consultas do usuário usam estruturas que manipulam um conjunto de dados sob a relação de ordem total. Entretanto, consultas que manipulam dados espaciais e métricos estão se tornando comuns entre os usuários que necessitam de suporte de uma estrutura de índice para respostas mais eficientes.

- Serialização de objetos: os atributos dos objetos são representados como um fluxo de dados serial. A maior vantagem da serialização é permitir que os dados naveguem entre diferentes dispositivos e ambientes operacionais.

- Não intrusivo: não realizar modificações nas classes criadas pelos usuários. Soluções intrusivas obrigam o modelo de classes do usuário a herdarem de uma classe persistente ou serem modificadas, com a adição de métodos, para realizar a persistência.

1.3 Organização do trabalho

Este documento está organizado da seguinte maneira:

- *Capítulo 2 - Métodos de Acesso*: discute sobre as principais técnicas de indexação, enfocando os métodos de acesso métricos e espaciais.

- *Capítulo 3 - Padrões em Frameworks*: apresenta os conceitos fundamentais envolvendo *frameworks*. Este Capítulo também discute os padrões de projeto (*Design Paterns*) utilizados neste trabalho.

- *Capítulo 4 - O Framework Object-Injection*: descreve o *framework* desenvolvido, mostrando seu funcionamento e detalhes de implementação.

- *Capítulo 5 - Experimentos*: apresenta os experimentos realizados com o *framework* desenvolvido, discutindo sua metodologia e resultados.

- *Capítulo 6 - Conclusão*: apresenta as conclusões do trabalho e propostas de metas futuras.

- *Apêndice A - A Métrica de Xu*: contém a descrição do cálculo de distância utilizado para indexar proteínas no espaço métrico.

Capítulo 2

Métodos de Acesso

Diversas aplicações computacionais estão centradas em torno de conjuntos de dados que, facilmente, superam o tamanho da memória primária. Um exemplo clássico é uma base de dados com múltiplas chaves de buscas, requerendo a habilidade de inserir, remover e consultar registros (SHAFFER, 2012). Para agilizar tais operações, sobretudo as consultas, muitas aplicações fazem uso de estruturas de índice para organizar seus registros de forma eficiente. Algumas aplicações utilizam apenas consultas simples, como "recuperar o registro cuja chave é k". Entretanto, outras aplicações requerem consultas mais elaboradas, como "localizar os n objetos mais parecidos com um objeto de consulta". Assim, diversos métodos de acesso de dados surgiram na literatura devido à essa necessidade por indexação e recuperação da informação.

Este Capítulo apresenta estruturas de armazenamento secundário usadas para organizar grandes coleções de objetos em disco. Tais estruturas suportam, eficientemente, operações de inserção, remoção e consulta, de maneira dinâmica. A Seção 2.1 introduz os métodos de acesso que lidam com objetos sujeitos à uma relação espacial. A Seção 2.2 discute as propriedades do espaço métrico e as estruturas de indexação para este espaço.

2.1 Métodos de acesso espaciais

Segundo Samet (1995), dados espaciais consistem de objetos espaciais, sejam eles pontos, retas, regiões, retângulos, superfícies, volumes ou até mesmo dados em mais dimensões, podendo incluir o tempo. Exemplos de dados espaciais são cidades, rios, estradas, países, plantações, montanhas etc. Exemplos de propriedades associadas a dados espaciais são a extensão de um rio, as fronteiras de um país etc.

A capacidade de responder à consultas espaciais baseia-se na forma em que os dados são representados e organizados. Embora todo Sistema Gerenciador de Banco de Dados (SGBD)

organize seus dados, ao lidar com dados espaciais a principal questão é *como* organizar estes dados. No caso dos dados espaciais, a organização deve ser baseada em suas chaves espaciais, ou seja, a indexação é realizada de acordo com o espaço ocupado pelo dado. Tais técnicas de indexação são conhecidas como *Métodos de Acesso Espaciais*, ou *Métodos de Acesso Multidimensionais* (SAMET, 1995). Estas técnicas foram originalmente desenvolvidas para indexar conjuntos de dados vetoriais, a fim de processar de forma eficiente consultas pontuais e por abrangência. Posteriormente, apareceram outras abordagens que estenderam as funcionalidades dos Métodos de Acesso Espaciais (SKOPAL, 2004a).

A seguir são apresentados a Árvore B (Seção 2.1.1) e o *Hash* Extensível (Seção 2.1.2), que são métodos de acesso unidimensionais e a Árvore R (Seção 2.1.3), como método de acesso multidimensional.

2.1.1 Árvore B

A bem conhecida Árvore B (BAYER; MCCREIGHT, 1972) (Figura 2.1) é uma estrutura de índices multi-níveis (GARCIA-MOLINA et al., 2008), balanceada em altura e dinâmica, usada para indexar conjuntos $\mathcal{S}$ de dados unidimensionais, em concordância com as seguintes propriedades (relação de ordem total):

- Transitividade: $\forall a, b, c \in \mathcal{S}, a < b \wedge b < c \Rightarrow a < c$.
- Anti-simetria: $\forall a, b \in \mathcal{S}, a < b \wedge a \neq b \Rightarrow \neg(b < a)$.
- Totalidade: $\forall a, b \in \mathcal{S} \Rightarrow a < b \vee b < a$.

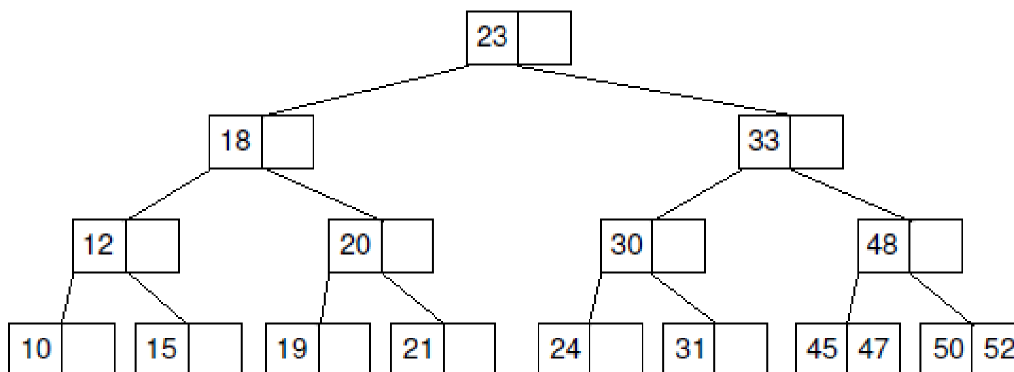


Figura 2.1: Representação de uma Árvore B

Fonte: Adaptada de Shaffer (2012)

A Árvore B armazena seus registros em dois tipos de nós: índices e folhas. Cada nó índice é constituído por M entradas e $M + 1$ ponteiros para os nós do nível imediatamente inferior, conforme a Figura 2.2. Os nós folha seguem a mesma estrutura, sendo formados por um

conjunto de M entradas, com a exceção de não possuírem apontadores para as sub-árvores. Cada entrada, seja de um índice ou de uma folha, é formada por um par (k, p), onde k representa a chave do objeto indexado pela Árvore B e p é a referência desse objeto na base de dados. Todas as entradas armazenadas em um nó encontram-se ordenadas.

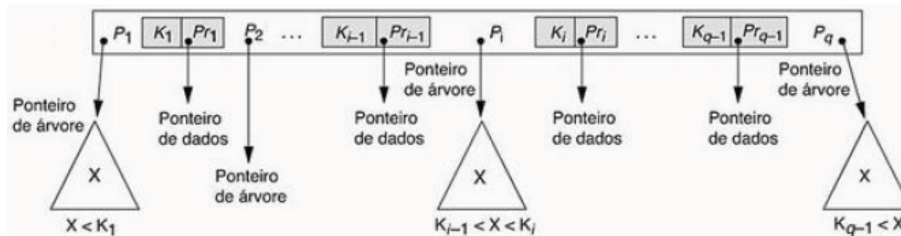


Figura 2.2: Nó índice de uma Árvore B
Fonte: Elmasri e Navathe (2011)

Os ponteiros dos nós índice são utilizados pelas operações de inserção e busca para localização de um registro. Conforme pode ser observado na Figura 2.2, uma chave k encontra-se localizada entre dois ponteiros, sendo que o da esquerda referencia a sub-árvore com chaves de valor menor que k, enquanto que o da direita referencia a sub-árvore cujos registros são maiores que k.

O algoritmo de busca da Árvore B, iniciado a partir do nó raiz, procura pela chave solicitada entre as entradas de cada nó, e, caso não encontre, inicia a busca recursivamente pela sub-árvore mais adequada à consulta. No momento em que a chave é encontrada, esteja ela em um índice ou folha, o algoritmo é interrompido.

O algoritmo de inserção é semelhante ao do processo de busca, exceto por percorrer a árvore da raiz até uma folha, que é onde as chaves são inseridas. Caso a folha selecionada não tenha espaço para acomodar a nova entrada (a folha já contém M entradas), ocorre uma quebra (*split*) do nó. Na operação de *split*, um novo nó é criado e as $M + 1$ entradas são divididas entre os dois nós, sendo que o elemento da posição $\lfloor (M + 1)/2 \rfloor$ (elemento central) é promovido a um nó índice; a nova folha armazenará as entradas maiores que o elemento central e a folha que causou o *split* fica com as entradas menores. O algoritmo de *split* garante uma taxa de ocupação mínima de, pelo menos, metade do nó.

A relação de ordem total, que restringe o conjunto de dados indexados pela Árvore B, pode ser suavizada para uma relação de ordem parcial. Nesse caso, seria possível inserir na árvore valores duplicados. Isto significa que objetos distintos de uma base de dados compartilham da mesma chave. Conforme descrito por Ramakrishnan e Gehrke (1999), o algoritmo de busca necessitaria de ser alterado de forma a devolver uma coleção de objetos caso uma duplicata fosse procurada. No entanto, o inconveniente das duplicatas persistiria na operação de remoção, sendo problemático localizar qual das duplicatas deveria ser excluída.

O trabalho de Comer (1979) introduziu uma variante especialmente interessante da Árvore B: a Árvore B+, mostrada na Figura 2.3.

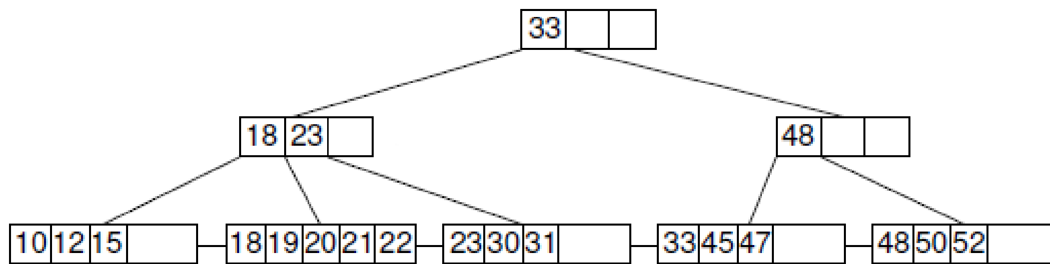


Figura 2.3: Representação de uma Árvore B+
Fonte: Adaptada de Shaffer (2012)

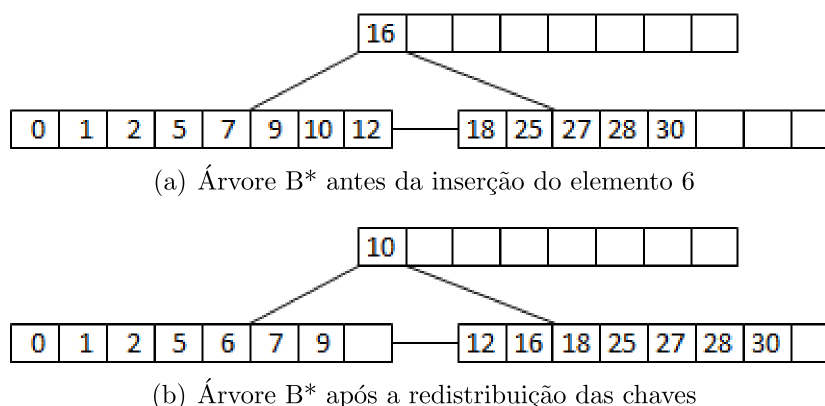
A Árvore B+ permite manter ordenado um conjunto de objetos e possibilita sua divisão em blocos de tamanho fixo, o que a torna ideal para uma estrutura de indexação baseada em disco. Enquanto que na Árvore B uma chave pode ser encontrada em algum de seus níveis, na Árvore B+ todas as chaves são armazenadas nas folhas, e somente as entradas em folha contêm a referência para o objeto na base de dados. As entradas dos nós índice são apenas objetos de referência, para direcionar buscas e inserções. Os nós folha da Árvore B+, como descrito por Elmasri e Navathe (2011), são interligados entre si para fornecer acesso sequencial ordenado no campo de pesquisa aos registros.

Cada nó folha da Árvore B+ tem a forma $(< k_1, p_1 >, ..., < k_M, p_M >, P_{prox})$, onde k_i é a chave representativa do objeto, p_i é a referência do objeto na base de dados e P_{prox} é o ponteiro para a próxima folha. Cada nó índice tem a forma $(ptr_1, k_1, ptr_2, k_2, ..., ptr_{M-1}, k_{M-1}, ptr_M)$, onde k_i é uma chave de referência e ptr_i é o ponteiro para uma sub-árvore (ELMASRI; NAVATHE, 2011).

O algoritmo de busca na Árvore B+ é idêntico ao da Árvore B, porém, ele não é interrompido ao encontrar uma entrada em índice correspondente à chave buscada; o processo se encerra apenas ao atingir o nível das folhas, que é onde está a referência do objeto.

A inserção na Árvore B+ também é idêntica a da Árvore B. A variação ocorre na operação de *split*, que, ao promover o objeto central, mantém uma cópia deste na nova folha.

Knuth (1998) propõe uma variação da Árvore B+, chamada Árvore B*. A Árvore B* é idêntica à B+, exceto pela política de *split*. Ao invés de dividir um nó ao meio (como ocorreria num *split* normal), suas entradas são redistribuídas com seu nó vizinho, conforme ilustra a Figura 2.4. Na Figura 2.4(a) tem-se a estrutura da Árvore B*, enquanto que a Figura 2.4(b) mostra a redistribuição das chaves após a inserção do elemento '6'. Somente se essa redistribuição não for possível, devidos aos nós vizinhos estarem cheios, ocorre o *split* propriamente dito. Na Árvore B*, a taxa de ocupação mínima de um nó é de 2/3 (quase 70%), ao invés de apenas 50%, como nas Árvores B e B+.



(a) Árvore B* antes da inserção do elemento 6

(b) Árvore B* após a redistribuição das chaves

Figura 2.4: Representação da Árvore B*

Fonte: Adaptada de Drozdek (2001)

2.1.2 Hash Extensível

O *Hash* Extensível (FAGIN et al., 1979) é uma estrutura de indexação dinâmica e orientada a disco que implementa um esquema de *hashing* (tabela de dispersão, espalhamento ou aleatorização). Esta estrutura é utilizada para responder à consultas exatas, i. e., encontrar um registro dado sua chave. Quando comparado com uma Árvore B+, que suporta consultas a um custo de $\mathcal{O}(\log_{k+1} n)$ operações de I/O (sendo k o maior número de chaves de um nó), o *Hash* Extensível apresenta um custo equivalente a $\mathcal{O}(1)$ operações (ZHANG et al., 2009).

Os nós do *Hash* Extensível são páginas de disco de tamanho fixo. Os nós folha, chamados de *buckets*, armazenam e organizam os dados em intervalos binários. O(s) nó(s) índice(s), conhecido(s) como *diretório(s)*, têm a forma de um *array* com 2^i entradas, onde cada entrada aponta para um *bucket*. É comum que mais de uma entrada de diretório aponte para o mesmo *bucket* (ZHANG et al., 2009). O valor i é chamado profundidade global do diretório. Cada *bucket* também tem um valor j, chamado de profundidade local. Esses valores são utilizados nas operações de inserção e busca.

Para determinar onde uma chave k será inserida o *Hash* Extensível utiliza os primeiros i bits da função *hash* $h(k)$, para escolher uma entrada no diretório. Acessando o *bucket* correspondente à entrada escolhida, os primeiros j bits são utilizados para determinar a posição onde a chave k será inserida. A Figura 2.5 ilustra em exemplo de *Hash* Extensível, onde o diretório tem profundidade 1. A entrada 0 do diretório aponta para um *bucket* onde as chaves começam por 0, enquanto que as chaves do *bucket* apontado por 1 têm seu primeiro bit igual a 1.

Supondo que fosse necessário inserir a chave 1010 no *Hash* da Figura 2.5, como seu primeiro bit é 1, a chave iria para o segundo *bucket*. Porém este *bucket* encontra-se cheio, devendo sofrer um *split*. Como as profundidades i e j têm o mesmo valor, o diretório dobra de tamanho, as chaves são reajustadas e a nova chave inserida, como mostra a Figura 2.6.

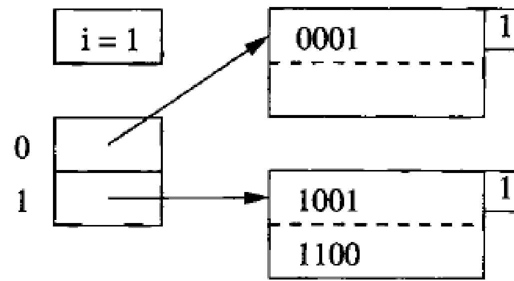


Figura 2.5: Exemplo de *Hash* Extensível
Fonte: Garcia-Molina et al. (2008)

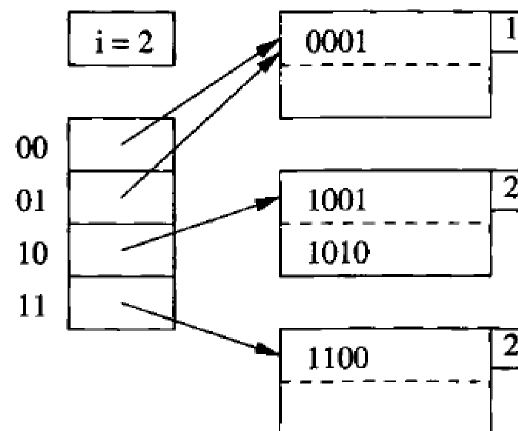


Figura 2.6: O diretório dobra após inserir 1010 na Figura 2.5.
Fonte: Garcia-Molina et al. (2008)

Fagin et al. (1979) explicam que, para um *Hash* Extensível com apenas 1 nível, são necessários apenas dois acessos a disco para recuperar qualquer objeto: um para o diretório e outro para o *bucket*. Isto é um pouco pior que o pior caso de uma estrutura de *hash* linear (1 acesso), mas muito melhor que o melhor caso de qualquer árvore ($\lg(n)$ acessos) (GAEDE; GÜNTHER, 1998).

A proposta de Fagin et al. (1979) inspirou ainda os trabalhos apresentados por Larson (1980) e Litwin (1980). Esses trabalhos discutem uma estrutura de *hash* linear que divide o espaço de endereçamento em intervalos binários, correspondentes aos *buckets*. Quando um *bucket* atinge sua capacidade, ele é dividido em dois *buckets*, sendo que cada um corresponde à metade do intervalo original do espaço de endereçamento. Uma outra proposta, Lomet (1983), apresenta uma estrutura muito parecida com o *Hash* Extensível original, exceto pelo fato de que o diretório é limitado a um determinado número de dobras. Ao atingir esse limite, ao invés de dobrar a estrutura do diretório, o *bucket* responsável por causar o *split* sofre a dobra, permitindo a inclusão de novas chaves.

2.1.3 Árvore R

A Árvore R (GUTTMAN, 1984) é uma extensão da Árvore B/B+ em um espaço multidimensional. Análoga à Árvore B, é uma estrutura de dados hierárquica, dinâmica e balanceada em altura. A Árvore R é utilizada para organizar dados espaciais de maneira eficiente, permitindo que sejam rapidamente recuperados, de acordo com suas localizações no espaço. Estes dados são objetos geométricos, de um conjunto n-dimensional, representados através de um retângulo delimitador mínimo (MBR - *Minimum bounding rectangle*), também n-dimensional. Um MBR pode se sobrepor a outro e/ou interceptar mais de um objeto, como ilustrado na Figura 2.7. Ainda assim, cada MBR está associado a um único objeto e incluído em um único nó da árvore, mesmo que isso não aconteça na representação geométrica.

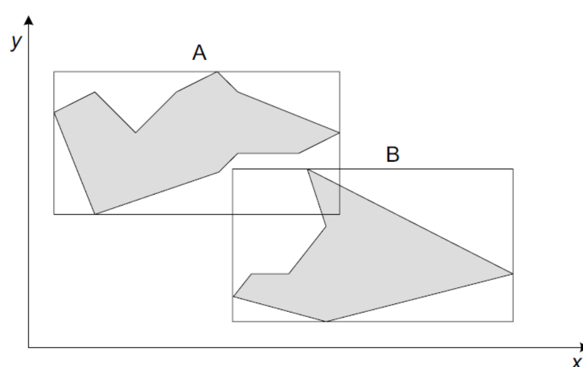


Figura 2.7: Exemplo de MBRs e objetos, onde os MBRs se interceptam mas os objetos não.
Fonte: Manolopoulos et al. (2006)

Os nós da Árvore R são páginas de disco, com uma taxa de ocupação de, pelo menos, metade do espaço total disponível. Cada nó corresponde a um MBR que delimita sua sub-árvore. As entradas das folhas contêm a referência do objeto no banco de dados, ao invés de ponteiros para uma sub-árvore. As principais características da Árvore R, reunidas por Guttman (1984), Gaede e Günther (1998) e Manolopoulos et al. (2006), são:

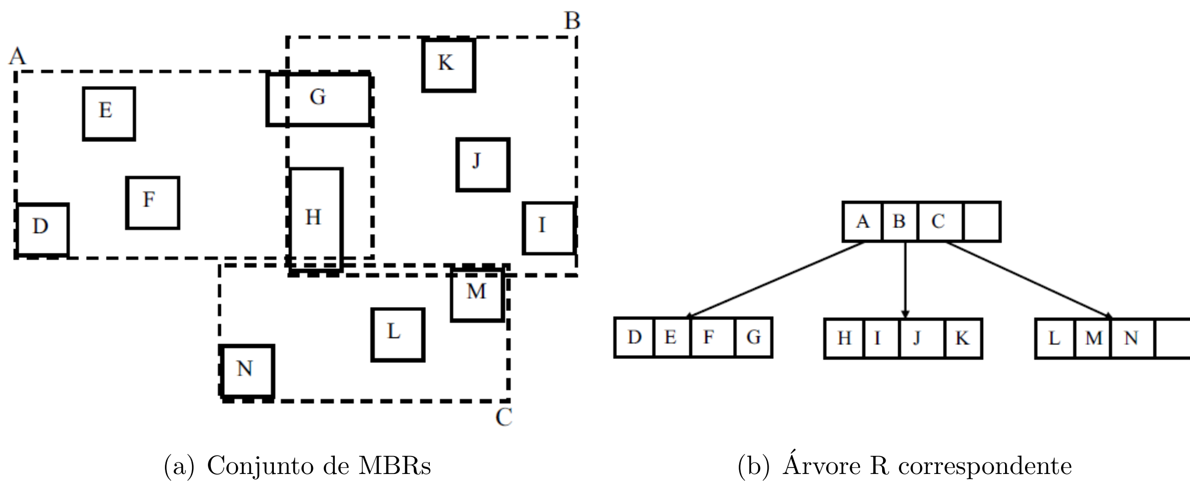
- Todo nó da árvore, seja folha ou índice, tem capacidade para M entradas. Caso o nó não seja a raiz, o número mínimo de entradas permitido é $m = \lfloor M/2 \rfloor$. O limite inferior m previne a degeneração da árvore e garante uma utilização eficiente do espaço de armazenamento. O limite superior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

- Cada entrada, em um nó folha, tem a forma (mbr, oid), onde mbr é o MBR que armazena o objeto e oid é o identificador do objeto.

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

- A raiz contém, no mínimo, duas entradas, exceto se for uma folha, podendo conter uma única entrada ou nenhuma, caso a árvore esteja vazia.
- Todas as folhas estão no mesmo nível.

A Figura 2.8(a) mostra um conjunto de MBRs bidimensionais de objetos de dados geométricos. Estes MBRs são D, E, F, G, H, I, J, K, L, M e N, que são armazenados no nível das folhas. A mesma figura mostra os três MBRs (A, B e C) que organizam, em nós internos, os retângulos mencionados. A Figura 2.8(b) ilustra a Árvore R correspondente ao conjunto de MBRs da Figura 2.8(a).



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

Figura 2.8: Representação da Árvore R
Fonte: Manolopoulos et al. (2006)

Cada representação distinta resultante de uma Árvore R original é determinada pela ordem de inserção (e/ou remoção) de suas entradas, de forma que muitas Árvores R podem representar o mesmo conjunto de dados retangulares (MANOLOPOULOS et al., 2006). O trabalho de Tzouramanis (2012), contudo, introduz uma variação da Árvore R, chamada de *History Independent R-Tree* (HIR-Tree), onde sua representação não é afetada pela ordem das inserções.

Conforme descrito por Guttman (1984), as operações de busca mais utilizadas em uma Árvore R são as consultas por abrangência (*range query*), que encontra todos os objetos que interceptam uma região de busca, e pontuais (*point query*), que faz o mesmo, mas com relação a um ponto do espaço. O algoritmo de busca percorre a árvore, a partir da raiz, de forma similar ao da Árvore B. A principal diferença é que, em cada nó índice, todas as entradas devem ser testadas para verificar se há alguma região de intersecção com a região de busca. Devido às intersecções 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 intersecção com a região/ponto buscado.

Como a Árvore R armazena somente MBRs, uma operação de consulta pode não devolver o objeto buscado (na Figura 2.7, uma região de busca pode pertencer a um MBR, mas não pertencer ao objeto.). Assim, uma consulta por abrangência devolve um conjunto de objetos candidatos, cuja extensão espacial deve ser testada com o espaço de busca, para verificar intersecções (GAEDE; GÜNTHER, 1998).

Roussopoulos et al. (1995) definem consultas aos vizinhos mais próximos (*k-nearest-neighbor queries*) através de um procedimento que evita a verificação de todas as entradas em um nó. Esta abordagem é baseada em métricas definidas no mesmo trabalho, que aferem valores mínimos e máximos de distância entre os objetos da Árvore R e um local de consulta.

O processo de inserção na Árvore R é similar à inserção numa Árvore B. Contrastando com o algoritmo de busca, apenas uma rota, da raiz até a folha, é seguida. A partir da raiz, até atingir uma folha, todas as entradas de um nó índice são examinadas e a sub-árvore escolhida é aquela cujo MBR sofre o menor aumento em sua área[1] para acomodar uma nova entrada. Empates são resolvidos escolhendo-se o MBR de menor área (GUTTMAN, 1984). Caso a folha selecionada não possua espaço suficiente para abrigar a nova entrada (a folha já contém M entradas), então as $M + 1$ entradas são divididas (*split*) entre a folha atual e uma nova folha. A operação de *split* em Árvores R é ligeiramente diferente da Árvore B, por considerar diferentes critérios (MANOLOPOULOS et al., 2006). Guttman (1984) propõe três diferentes políticas de divisão de um nó e promoção de objetos:

- **Divisão Linear**: possuindo complexidade temporal proporcional a $\mathcal{O}(M)$, escolhe, entre as $M + 1$ entradas, dois objetos que estejam o mais distante possível no espaço como primeiros elementos de cada um dos dois nós. Distribui as $M-1$ entradas restantes entre os dois nós, respeitando o critério do menor aumento do MBR.
- **Divisão Quadrática**: com complexidade temporal de $\mathcal{O}(M^2)$, o algoritmo seleciona duas das $M + 1$ entradas para serem os primeiros elementos dos dois novos nós. São selecionadas as entradas que consomem a maior área se ambas fossem colocadas no mesmo nó, isto é, a área do retângulo cobrindo ambas as entradas menos a área dessas entradas (*dead space*) deve ser a maior possível. Até que as $M - 1$ entradas tenham sido distribuídas, seleciona, uma a uma, aquela cuja diferença de *dead space* para cada um dos dois nós é máxima, e a adiciona no nó cujo MBR sofre o menor aumento.
- **Divisão exaustiva**: com complexidade temporal de $\mathcal{O}(2^{M-1})$, testa todos os possíveis pares de objetos, selecionando o melhor deles, com relação ao mínimo aumento do MBR.

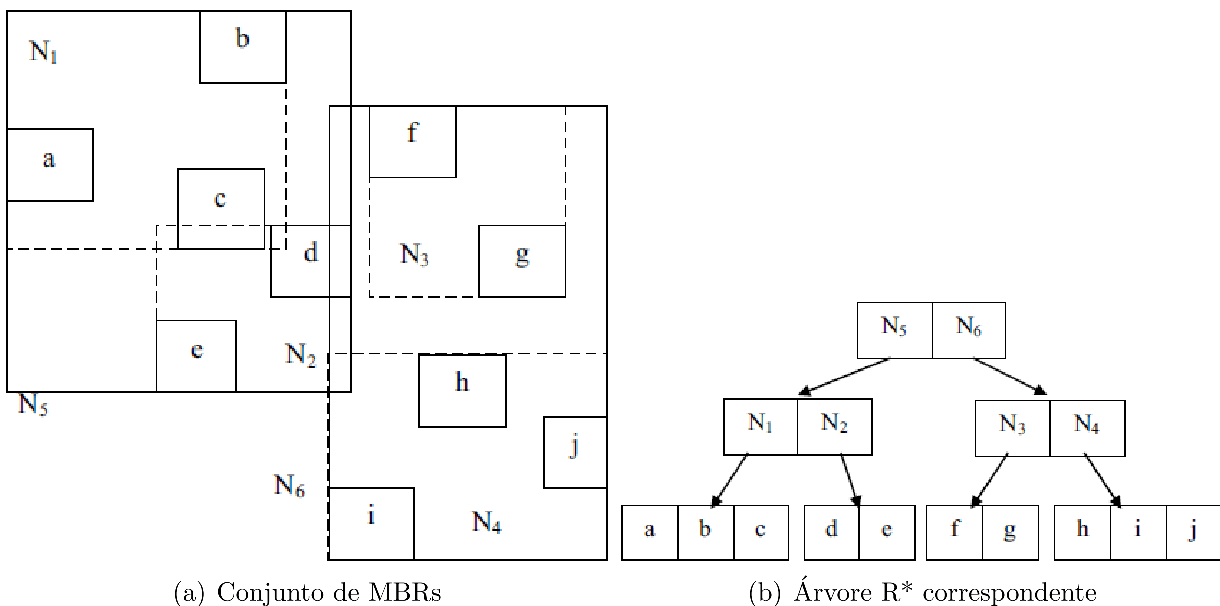
Gaede e Günther (1998) apontam algumas variantes da Árvore R, como a Árvore R+, a Árvore R de Hilbert e a Árvore R*.

[1] Área no caso de um conjunto de dados bidimensionais. Para mais dimensões, considera-se o volume ou hiper-volume.

A Árvore R+ (STONEBRAKER et al., 1986; SELLIS et al., 1987) melhora o mecanismo de busca da Árvore R original, evitando a necessidade de descer em mais de um ramo. Para isso, a inserção na Árvore R+ é feita de forma a não permitir sobreposição de MBRs no mesmo nível, inserindo a mesma entrada em mais de um nó, caso haja intersecção entre os nós e a entrada.

A Árvore R de Hilbert (KAMEL; FALOUTSOS, 1994) é uma estrutura híbrida, que combina as propriedades da Árvore R original e da Árvore B+. Seus objetos são caracterizados por um valor numérico, chamado valor de Hilbert, calculado a partir seus centroides. Estes centroides são inseridos em uma Árvore B+.

A Árvore R* (BECKMANN et al., 1990) introduz melhorias nos algoritmos da Árvore R original, principalmente com relação às operações de inserção e *split*. Seus autores reforçam a ideia de que o processo de inserção é crítico para um bom desempenho das consultas. Dessa forma, é proposta uma alteração na política de inserção, a 'reinserção forçada': ao invés de sofrer um *split*, algumas entradas de um nó cheio são removidas e reinseridas na árvore. Outra questão investigada neste trabalho é uma alteração na política de *split*, onde o principal foco não é reduzir a área dos MBRs, mas reduzir a sobreposição entre os nós de um mesmo nível (GAEDE; GÜNTHER, 1998). A Figura 2.9 mostra a representação da Árvore R*.



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

Figura 2.9: Representação da Árvore R*
Fonte: Manolopoulos et al. (2006)

Investigações sobre a importância dos processos de inserção e *split* também aparecem no trabalho de Brakatsoulas et al. (2002). Este trabalho compara a construção da Árvore R a um problema de *clustering*, ou agrupamento. Para isso, os autores propõem a utilização do bem-conhecido algoritmo *k-means* nas operações de *split*.

2.2 Métodos de acesso métricos

Com o crescimento dos dados digitais, tanto em extensão quanto em variedade, repositórios cada vez maiores são necessários para fins de armazenamento. Para que os usuários sejam capazes de acessar um determinado dado, esses objetos devem ser estruturados e manipulados de maneira eficaz (ZEZULA et al., 2006). Contudo, principalmente com relação a dados multimídia, os repositórios de dados tornam-se desestruturados, pois novos tipos de dados tais como textos, imagens, áudio e vídeo podem ser armazenados e buscados, não sendo possível organizar a informação em chaves e registros (CHÁVEZ et al., 2001), como nos SGBDs tradicionais. Tamanha é a variedade entre esses objetos que eles não podem nem mesmo ser comparados através de operadores relacionais usuais ($=, \neq, <, ...$) (FALOUTSOS, 1996). Dessa forma, o único critério de comparação entre essas informações é uma medida do quão parecido um objeto é do outro. À busca por objetos seguindo esse critério de comparação dá-se o nome de *busca por similaridade*.

A busca por similaridade é uma técnica de recuperação da informação onde, dado um objeto de consulta, o resultado desejado é um conjunto de objetos considerados similares ao objeto consultado (HETLAND, 2009). Segundo Zezula et al. (2006), o critério de similaridade é formalizado por uma *função de distância*, também chamada de *métrica*. Sendo $\mathcal{D}$ um domínio de objetos e d uma métrica sobre $\mathcal{D}$, um *espaço métrico* é definido por $\mathcal{M} = (\mathcal{D}, d)$. Em um espaço métrico, as propriedades da função $d : \mathcal{D} \times \mathcal{D} \mapsto \mathbb{R}$ são caracterizadas por:

- Não-negatividade: $\forall x, y \in \mathcal{D} \Rightarrow d(x, y) \geq 0$
- Simetria: $\forall x, y \in \mathcal{D} \Rightarrow d(x, y) = d(y, x)$
- Reflexividade: $\forall x \in \mathcal{D} \Rightarrow d(x, x) = 0$
- Positividade: $\forall x, y \in \mathcal{D}, x \neq y \Rightarrow d(x, y) > 0$
- Desigualdade triangular: $\forall x, y, z \in \mathcal{D} \Rightarrow d(x, y) \leq d(x, z) + d(y, z)$

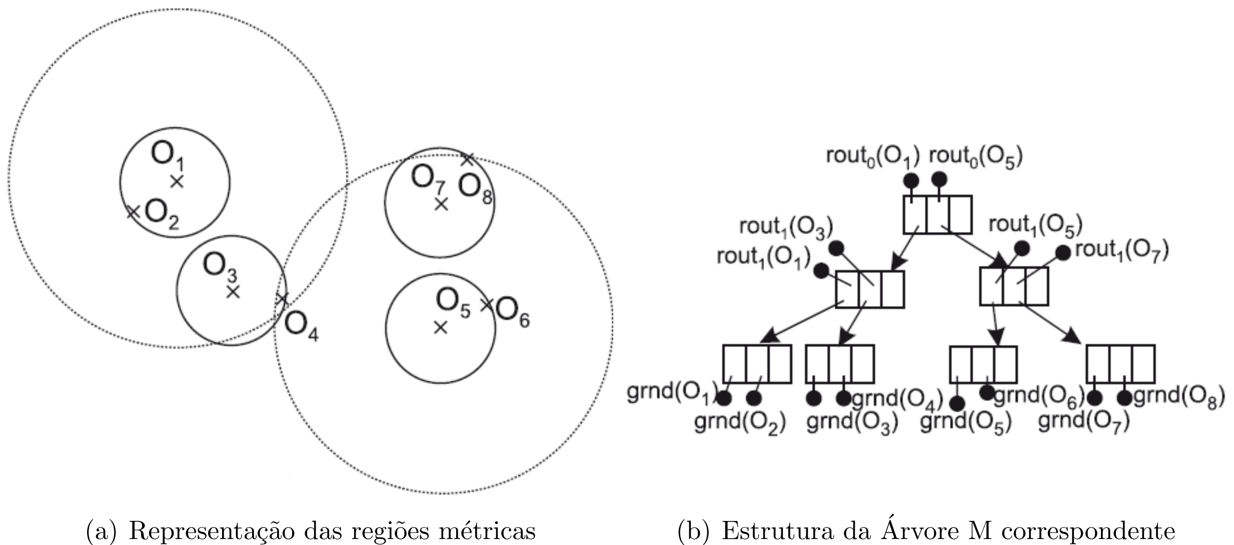
As estruturas apresentadas a seguir, Árvore M (Seção 2.2.1) e Árvore Slim (Seção 2.2.2, são chamados de Métodos de Acesso Métricos. Através da utilização de uma métrica, elas conseguem indexar um conjunto de objetos de maneira eficiente, adequada à consultas.

2.2.1 Árvore M

A Árvore M, introduzida por Ciaccia et al. (1997) e estendida por Patella (1999), é uma árvore híbrida, baseada nas Árvores B/B+ e R*, que foi inicialmente projetada para permitir consultas por similaridade em bases de dados multimídia (SKOPAL et al., 2003; SKOPAL; LOKOČ,

2009). Trata-se de uma estrutura de dados dinâmica, hierárquica, orientada a armazenamento secundário e balanceada em altura. A Árvore M é utilizada para indexar objetos de conjuntos de dados métricos, que não seguem uma relação de ordem total, relacionando-se apenas por sua similaridade ou dissimilaridade (ZEZULA et al., 2006).

De maneira equivalente à Árvore R, os nós da Árvore M são páginas de disco de tamanho fixo. Os nós folha armazenam os objetos indexados na base de dados, representados por suas chaves, enquanto que os nós índice armazenam os chamados objetos de roteamento (CIACCIA et al., 1997). Objetos de roteamento, definidos pela política de *split*, fornecem rotas para atingir nós folha. Cada objeto, em um nó folha, tem a forma (o_i, oid, d), onde o_i é a chave do objeto da base de dados, oid é o identificador do objeto e d é a distância de o_i para seu objeto representativo o_p. Cada entrada de um nó índice (interno ou de rota) tem a forma (o_j, ptr, r_j, d), onde o_j é o objeto representante de uma região do espaço, ptr é o apontador para a sub-árvore correspondente, r_j é o raio de cobertura da região centrada em o_j e d é a distância de o_j para seu objeto representativo o_r. A Figura 2.10 ilustra uma Árvore M. Na Figura 2.10(a), tem-se a divisão do espaço métrico em regiões circulares; enquanto que a Figura 2.10(b) mostra a representação correspondente dos nós da Árvore M, onde 'rout' (*routing*) são os objetos de roteamento e 'grnd' (*ground*) são os objetos nas folhas.

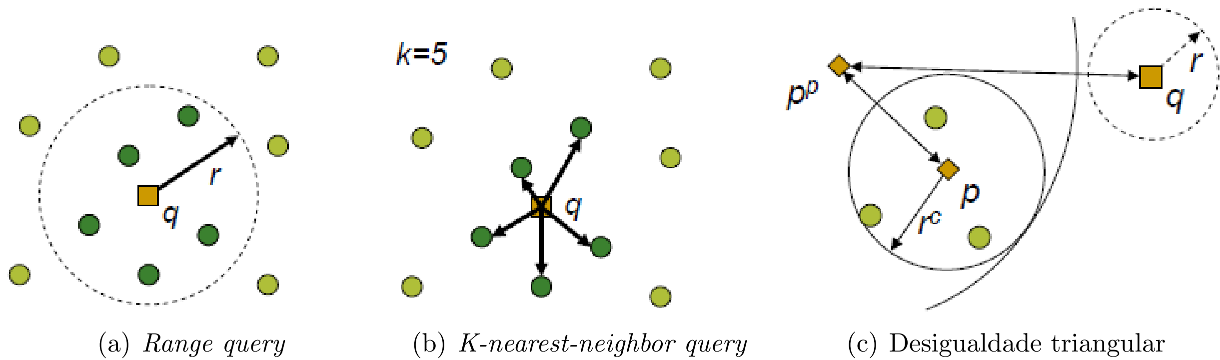


(a) Representação das regiões métricas (b) Estrutura da Árvore M correspondente

Figura 2.10: Representação da Árvore M
Fonte: Skopal (2006)

Do ponto de vista semântico, o raio de cobertura r_i delimita uma região do espaço métrico centrada em um objeto o_i qualquer. Dessa forma, todos os objetos o_j contidos nessa região são cobertos pela distância r_i, isto é, $d(o_i, o_j) \leq r_i$.

Concebida para responder a buscas por similaridade, as consultas por abrangência (*range queries* - Figura 2.11(a)) e pelos k-vizinhos mais próximos (k-*nearest-neighbors* - Figura 2.11(b)) são as operações de busca mais comuns na Árvore M (LOKOČ, 2008).



(a) *Range query* (b) *K-nearest-neighbor query* (c) Desigualdade triangular

Figura 2.11: Consultas na Árvore M
Fonte: Zezula et al. (2006)

Dados um objeto q e um raio de busca r a partir de q, uma consulta por abrangência inicia-se pelo nó raiz e percorre os ramos da árvore recursivamente, excluindo do processo de busca as sub-árvores em que $|d(q,p) - d(p,p^p)| - r^c > r$ (desigualdade triangular), como mostrado na Figura 2.11(c) (CIACCIA et al., 1997; ZEZULA et al., 2006). Ainda segundo Ciaccia et al. (1997), ao acessar um determinado nó N, como a distância entre q e N^p (objeto representativo do nó N) já foi computada, é possível realizar podas nas sub-árvores de N sem a necessidade de recalcular distâncias para todos os outros objetos.

Assumindo que, pelo menos, k objetos estejam armazenados na Árvore M, o algoritmo de consulta aos k vizinhos mais próximos, similar ao da Árvore R, assemelha-se ao algoritmo de busca por abrangência. Entretanto, o critério de poda da busca aos vizinhos mais próximos é dinâmico, isto é, o raio de busca é definido pela distância entre o objeto de busca q e o k-ésimo objeto mais próximo, em um dado momento. Como descrito por Patella (1999), se cerca de k objetos, muito próximos a q, são encontrados durante as primeiras verificações do processo de busca, uma quantidade maior de sub-árvores será podada, aumentando significativamente o desempenho.

O algoritmo de inserção na Árvore M, de forma análoga ao das Árvores B/B+ e R, desce os níveis da árvore recursivamente, a partir da raiz, a fim de encontrar o nó folha mais apropriado para acomodar a nova entrada o_n. O conceito usado para localizar o 'nó mais apropriado' é descer pela sub-árvore em que nenhum aumento do raio de cobertura é necessário, isto é, a cada nível, é selecionada a sub-árvore em que $d(o_i, o_n) \leq r_i$. Caso mais de uma entrada satisfaça esta propriedade, a sub-árvore escolhida é aquela em que a distância $d(o_i, o_n)$ é mínima. Se o empate ainda persistir, seleciona-se a sub-árvore da entrada de mínimo raio r_i. Novos empates são resolvidos escolhendo-se qualquer uma das sub-árvores empatadas, aleatoriamente.

Caso não haja um objeto de roteamento que possa conter a nova entrada o_n, ou seja, $\forall i \Rightarrow d(o_i, o_n) > r_i$, um dos objetos contidos no nó deve sofrer um aumento em sua cobertura.

Dessa forma, é escolhida a sub-árvore em que o aumento do raio de cobertura seja o menor possível, isto é, minimiza-se o valor de $d(o_i, o_n) - r_i$.

Como toda estrutura de dados balanceada, em que os objetos são inseridos em folhas, pode acontecer de não haver espaço suficiente para armazenar uma nova entrada em uma determinada folha. Nesse caso, ocorre uma divisão do nó que está cheio, através de duas operações: *promote* e *split*. Durante este processo, um novo nó é criado e o conjunto de entradas do nó cheio mais o novo objeto a ser inserido ($N+1$ objetos) são redistribuídos entre esses dois nós. Posteriormente, dois objetos, um representante de cada um dos dois nós, são promovidos ao nível superior.

A operação de promoção (*promote*) é responsável por escolher dois elementos, entre as $N+1$ entradas, para serem os objetos de roteamento, armazenados no nó índice representativo. A escolha dos objetos promovidos tem impacto direto, positivo ou negativo, nas operações de consulta, pois afetam o raio de cobertura das entradas. Alguns dos possíveis critérios para escolha dos objetos p_1 e p_2 a serem promovidos, apresentados por Ciaccia et al. (1997) e Zezula et al. (2006), são:

- *Random*: escolhe p_1 e p_2 aleatoriamente. Este é o algoritmo mais rápido, porém com impactos geralmente negativos sobre a busca.
- *mRad* (*minimum radius*): escolhe p_1 e p_2 com mínimo valor de $(r_1 + r_2)$.
- *mMRad* (*min-max radius*): escolhe p_1 e p_2 com mínimo valor de $max(r_1, r_2)$. Apontado por Ciaccia et al. (1997), Patella (1999) e Skopal (2004a) como o que produz a árvore mais otimizada para a operação de busca, porém com consumo de tempo quadrático.
- *mLB-Distance* (*minimum lower bound distance*): definido um objeto p_1, escolhe p_2 tal que $d(p_1, p_2)$ seja máxima.

Uma vez definidos os objetos de roteamento, o algoritmo de quebra (*split*) é responsável por criar um novo nó e distribuir as $N + 1$ entradas entre os nós 'cheio' e novo, de acordo com um dos seguintes critérios (CIACCIA et al., 1997):

- **Hiperplano generalizado**: para cada objeto o_j, se $d(p_1, o_j) \leq d(p_2, o_j)$, colocar o_j no nó $\mathcal{N}_{cheio}$, senão, atribuir o_j ao nó $\mathcal{N}_{novo}$.
- **Divisão balanceada**: para cada objeto o_j, calcular $d(p_1, o_j)$ e $d(p_2, o_j)$. Até que todas as entradas tenham sido distribuídas, repetir os seguintes passos:
  - Atribuir ao nó $\mathcal{N}_{cheio}$ a entrada o_j de menor distância para p_1.
  - Atribuir ao nó $\mathcal{N}_{novo}$ a entrada o_j de menor distância para p_2.

Embora a divisão balanceada seja computacionalmente mais cara (tempo proporcional a $\mathcal{O}(n^2)$, onde n é o número de entradas) que a distribuição através do hiperplano ($\mathcal{O}(n)$), ela garante a taxa de ocupação de pelo menos metade do nó, o que contribui para uma árvore de menor altura.

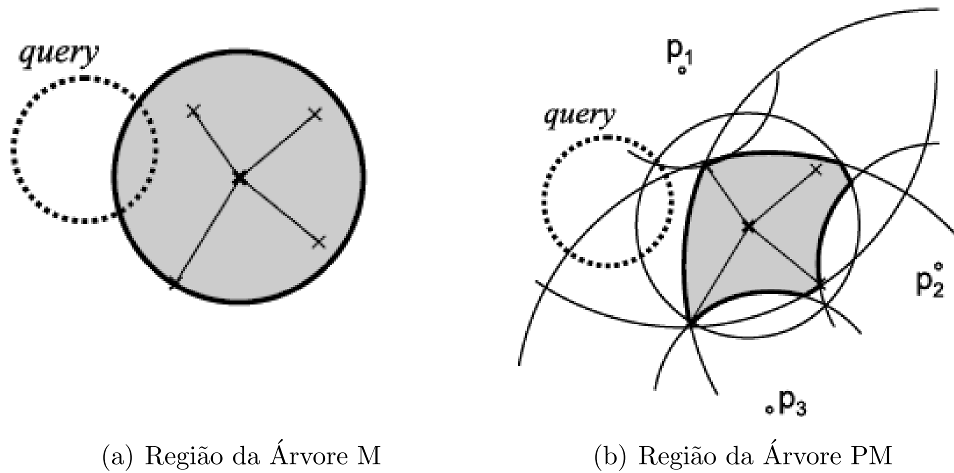
Patella (1999) descreve a política de *split* como uma etapa essencial, que deve ser cuidadosamente projetada para gerenciar, de forma eficaz, a evolução da Árvore M. Particularmente, toda política de *split* deveria seguir os seguintes requisitos, apontados pelo autor:

- **Cobertura mínima**: como as podas no espaço de busca são baseadas no raio de cobertura dos objetos de roteamento, este deve ser mantido tão pequeno quanto seja possível.
- **Sobreposição mínima**: para reduzir o número de caminhos que devem ser atravessados para responder a uma consulta, é conveniente que as regiões de cobertura dos objetos de roteamento estejam bem separadas.
- **Balanceamento**: este é um requisito básico de qualquer estrutura de dados balanceada em altura, garantindo que a altura da árvore seja a menor possível.
- **Tempo de CPU**: outro fator que deve ser considerado é o custo para computar as distâncias entre os objetos. Uma vez que este cálculo é realizado inúmeras vezes, algoritmos quadráticos ou ainda mais complexos podem tornar-se impraticáveis para determinadas aplicações.

Similar à Árvore R, algumas variantes da Árvore M surgiram na literatura. O trabalho de Ciaccia e Patella (2000) estende o princípio da Árvore M em uma estrutura de dados adequada à recuperação de conteúdo proveniente de múltiplas bases de dados. Esta estrutura, nomeada Árvore M^2, indexa múltiplos espaços métricos e seus objetos, usando mais de uma métrica ao mesmo tempo. Segundo os autores, a Árvore M^2 é adequada para objetos que são representados por múltiplas características.

A Árvore M+ (ZHOU et al., 2003) introduz o conceito de *dimensão de chave*, através da qual um espaço métrico é dividido em dois sub-espaços, chamados de *espaços gêmeos*, que não se sobrepõem. Dessa forma, as operações de consulta apresentam um desempenho superior ao da Árvore M original. Além disso, os algoritmos de construção da Árvore M+ foram projetados de forma a otimizar o espaço de armazenamento dentro do nó.

A Árvore PM (*Pivoting M-Tree*) (SKOPAL et al., 2004; SKOPAL, 2004b) particiona o espaço métrico em regiões formadas pela intersecção de hiper-esferas e hiper-anéis. Os hiper-anéis são definidos por pontos fixos, chamados de *pivôs*. Como conseqüência, os objetos indexados são mais fortemente restritos ao formato da região, o que aumenta o desempenho das operações de busca. A Figura 2.12 ilustra regiões do espaço métrico definidas pelas Árvores M e PM (com 3 pivôs).



(a) Região da Árvore M (b) Região da Árvore PM

Figura 2.12: Regiões do espaço métrico nas Árvores M e PM
Fonte: Skopal (2004b)

2.2.2 Árvore Slim

A Árvore Slim (TRAINA JR. et al., 2000) é uma estrutura de dados dinâmica, balanceada em altura e utilizada para indexar objetos de um espaço métrico. A Árvore Slim compartilha da mesma estrutura básica de outras árvores métricas, como a Árvore M, onde os dados são armazenados nas folhas e uma hierarquia de agrupamento é construída nos índices. Ainda segundo Traina Jr. et al. (2000), as diferenças entre a Árvore Slim para outros métodos de acesso métricos são: um novo algoritmo de inserção, alterações na política de *split* e a introdução de um novo algoritmo, o *Slim-Down*, utilizado para reduzir a sobreposição entre as regiões do espaço métrico.

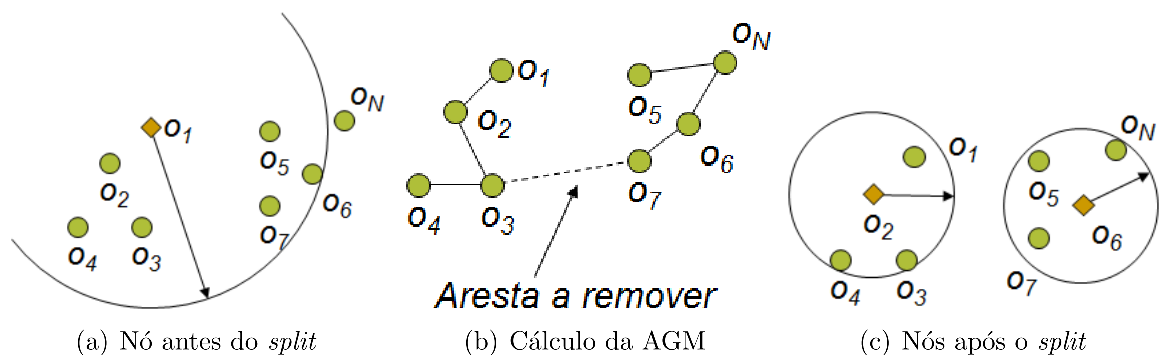
Os nós da Árvore Slim são páginas de disco de tamanho fixo. Nós do tipo folha contêm todos os objetos armazenados na árvore, representados por suas chaves. Cada objeto, em folha, tem a forma (o_i, oid, d), onde o_i é a chave do objeto na base de dados, oid é o identificador do objeto e d é a distância de o_i para seu objeto representativo o_p. De forma análoga à outras árvores, os nós índice armazenam os objetos que formam a rota para os nós folha. Cada entrada de um nó índice tem a forma (o_j, ptr, r_j, d, ne) , onde o_j é o objeto representante de uma região do espaço, ptr é o apontador para a sub-árvore correspondente, r_j é o raio de cobertura da região centrada em o_j, d é a distância de o_j para seu objeto representativo o_r e ne é o número de entradas do nó referenciado por ptr (TRAINA JR. et al., 2002).

A inserção na Árvore Slim desce a árvore de forma recursiva, a partir da raiz, localizando o nó mais apropriado para cobrir o novo objeto o_n. Enquanto que outras árvores métricas escolhem como objeto representativo de o_n aquele que possui a menor distância $d(o_r, o_n)$, ou, se não houver cobertura, aquele cujo raio teria a menor expansão, a Árvore Slim oferece três algoritmos de escolha da sub-árvore:

- *Random*: de forma aleatória, escolhe um dos objetos que se qualificam como objetos representativos para a nova chave.
- *minDist*: escolhe o nó que tem a menor distância do novo objeto para o seu representativo.
- *minOccup*: dentre os nós que se qualificam como sub-árvore, escolhe aquele que tem o menor número de entradas. Como mostrado por Traina Jr. et al. (2000) e Traina Jr. et al. (2002), este é o método que garante melhor desempenho nas consultas. O número de entradas de cada nó é armazenado no campo *ne*, como descrito anteriormente.

Caso um dado nó não contenha espaço suficiente para acomodar uma nova entrada, ele sofre um *split*: cria-se um novo nó e as entradas são distribuídas. Para escolha dos objetos que serão promovidos ao nível superior, Traina Jr. et al. (2000) apresentam três algoritmos:

- *Random*: de forma aleatória, escolhe um dos objetos que se qualificam como objetos representativos para a nova chave.
- *minMax*: escolhe dois objetos com mínimo valor de $max(r_1, r_2)$. Idêntico ao algoritmo *mMRad* da Árvore M, apresentado na Seção 2.2.1.
- *Minimal Spanning Tree*: calcula a Árvore Geradora Mínima (AGM (KRUSKAL, 1956)) formada pelas entradas a serem redistribuídas e removendo a maior aresta. Como resultado, tem-se as entradas divididas e distribuídas entre dois conjuntos. Como objetos representativos desses conjuntos, escolhe-se as entradas de menor distância para as demais (TRAINA JR. et al., 2000). A Figura 2.13 ilustra o processo de *split*: na Figura 2.13(a) tem-se o nó que será dividido; a Figura 2.13(b) ilustra o cálculo da Árvore Geradora Mínima e a aresta que será removida; na Figura 2.13(c) tem-se a nova distribuição das entradas, após o *split*.

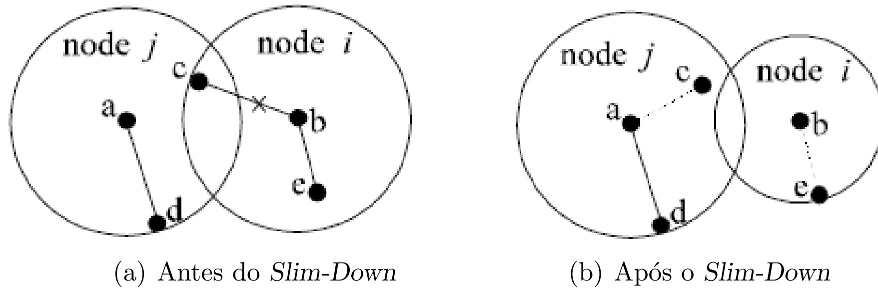


(a) Nó antes do *split* (b) Cálculo da AGM (c) Nós após o *split*

Figura 2.13: Exemplo de *split* na Árvore Slim

Fonte: Adaptada de Zezula et al. (2007)

O algoritmo *Slim-Down* pode ser aplicado na Árvore Slim para melhorar a qualidade das operações de busca, reorganizando a árvore após as inserções. A ideia do algoritmo é procurar nos nós folha pelo objeto mais distante do objeto representativo (em índice) e tentar encontrar uma folha mais apropriada. Se essa folha existir, o objeto é inserido nela (sem alterações na cobertura) e removido da folha antiga, cujo valor do raio é reduzido (TRAINA JR. et al., 2000). A Figura 2.14 mostra a aplicação do algoritmo *Slim-Down*, onde o objeto 'c', antes pertencente ao nó i, é realocado no nó j.

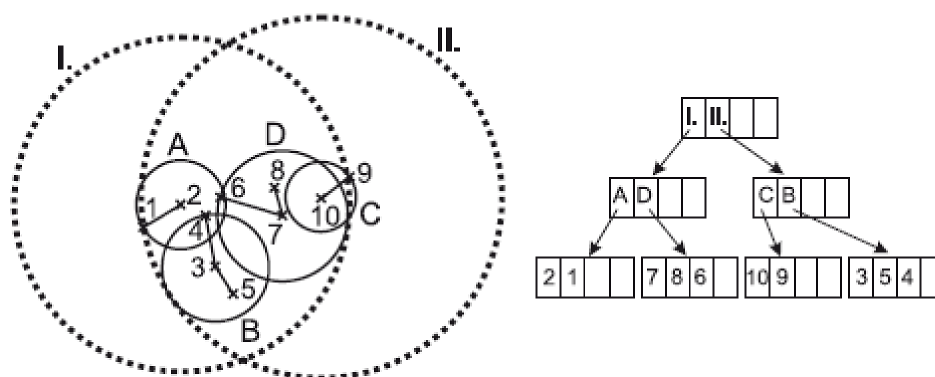


(a) Antes do *Slim-Down* (b) Após o *Slim-Down*

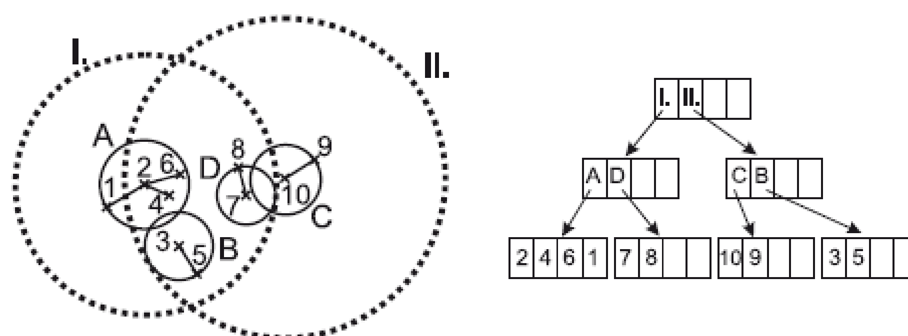
Figura 2.14: Aplicação do algoritmo *Slim-Down*
Fonte: Traina Jr. et al. (2000)

Baseando-se no algoritmo *Slim-Down* original, o trabalho de Skopal et al. (2003) introduziu uma variação do mesmo, chamado *Generalized Slim-Down*, que pode ser aplicado às entradas dos nós índice. A ideia do algoritmo *Generalized Slim-Down* é percorrer separadamente todos os níveis da árvore, iniciando pelas folhas, e, para cada nó N de um dado nível, tentar encontrar uma melhor localização para seus elementos. Para cada entrada o_i, em um nó folha N, o algoritmo executa uma *range query* com raio zero sobre o_i, a fim de recuperar um conjunto de nós relevantes (de melhor localização) para o_i. De forma similar, se o nó N analisado for um índice, para cada entrada o_i, o algoritmo executa uma *range query* sobre o_i, sendo que o raio é a cobertura de o_i. A partir do conjunto de nós que cobrem inteiramente a região definida pela *range query*, o algoritmo busca um nó M cujo representativo de seu nó pai esteja mais próximo de o_i que o representativo de o_i. Se tal nó M existir e contiver espaço suficiente, a entrada o_i é movida de N para M e o raio de cobertura de N pode ser reduzido. O algoritmo persiste até atingir o nó raiz.

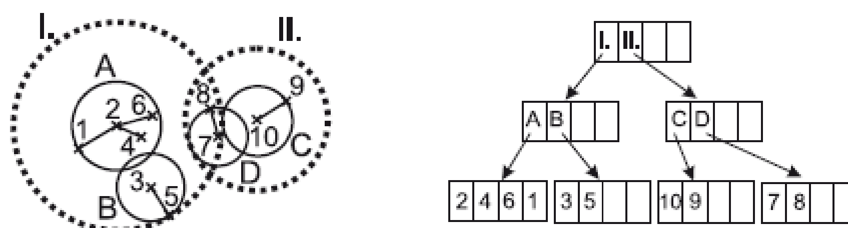
A Figura 2.15 ilustra o funcionamento do algoritmo *Generalized Slim-Down*. Na Figura 2.15(a), tem-se a árvore original, após as inserções dos objetos. A Figura 2.15(b) mostra a árvore resultante após a aplicação do algoritmo no nível das folhas. Os objetos '4' e '6', antes pertencentes à uma região de sobreposição, foram remanejados, possibilitando a redução do raio de cobertura das regiões 'B', 'D' e 'II'. A Figura 2.15(c) mostra o resultado do algoritmo no nível imediatamente superior às folhas. As entradas 'B' e 'D' também foram movidas, possibilitando a redução da cobertura das regiões 'I' e 'II'. Na iteração seguinte, o algoritmo encerra, pois atinge o nó raiz.



(a) Árvore após as inserções



(b) Aplicação do algoritmo *Generalized Slim-Down* nas folhas



(c) Aplicação do algoritmo *Generalized Slim-Down* no nível intermediário

Figura 2.15: Aplicação do algoritmo *Generalized Slim-Down*
Fonte: Skopal (2004a)

2.3 Considerações finais

Este Capítulo apresentou alguns métodos de acesso multidimensionais e métricos. Como métodos de acesso espaciais, foram apresentados a Árvore B e o *Hash* Extensível, que são unidimensionais e a Árvore R, multidimensional. Estes métodos são consagrados na literatura por responderem com eficiência à consultas dependentes da geometria de seus objetos ou sujeitas a uma relação de ordenação dos mesmos.

Os métodos de acesso métricos baseiam-se em uma relação de distância entre objetos, para realizar operações de inserção e consulta. Como representantes deste tipo de método de acesso, foram apresentadas a Árvore M e a Árvore Slim. Entre os principais tipos de consultas métricas, destacam-se a consulta pontual, por abrangência e pelos vizinhos mais próximos.

Os aspectos discutidos neste Capítulo constituem o embasamento teórico utilizado na implementação do *framework* proposto no Capítulo 4.

Capítulo 3

Padrões em *Frameworks*

Frameworks e padrões de projeto (*Design Patterns*) são largamente aplicados não só no desenvolvimento de *software* reutilizáveis mas também na análise, modelagem e *debug* dos mesmos. Este Capítulo introduz os conceitos relativos a *frameworks* e padrões de projeto, apresentando também as vantagens de sua utilização. A Seção 3.1 discute sobre as definições de *framework* e suas diversas classificações e benefícios. Esta Seção encerra fazendo uma distinção entre o que é um *framework*, uma aplicação, uma biblioteca de classes (*library*) e um padrão de projeto. A Seção 3.2 apresenta os conceitos envolvendo os padrões de projeto utilizados no *framework* proposto neste trabalho, mostrando também exemplos práticos da aplicação desses padrões.

3.1 *Frameworks*

Entre as principais definições de *framework* presentes na literatura, destacam-se:

- Um *framework* é uma coleção de classes que constituem uma aplicação abstrata para a solução de um conjunto de problemas (FAYAD; SCHMIDT, 1997).
- Um *framework* é um conjunto de classes cooperantes que constroem um projeto reutilizável para uma determinada categoria de *software* (GAMMA et al., 1995; DEUTSCH, 1989).
- Um *framework* é uma aplicação reutilizável e parcialmente completa que pode ser especializada para construir aplicações customizadas (JOHNSON; FOOTE, 1988).
- Um *framework* é um projeto arquitetural de um sistema que descreve seus componentes e como eles interagem (CAMPBELL et al., 1992).

Embora distintas, as definições apresentadas, além de não serem contraditórias, completam-se mutuamente.

De acordo com Mattsson e Bosch (2000), *frameworks* fornecem uma solução estruturada para um conjunto de problemas contendo pontos fixos (*frozen spots*) e pontos flexíveis (*hot spots*). Os pontos fixos definem um conjunto de funcionalidades padrão que já foram implementadas e são fornecidas na arquitetura do *framework*. Os pontos flexíveis são os componentes que podem ser customizados e estendidos de acordo com a necessidade e individualidade de cada aplicação. Essa customização é feita através da implementação de interfaces, extensão de classes abstratas e configuração de classes existentes. Dessa forma, as aplicações são instanciadas a partir do reuso da arquitetura definida pelo *framework* e da personalização de seus pontos flexíveis (MATTSSON, 2000).

Fayad e Schmidt (1997), Fayad et al. (1999), Mattsson (2000) e Mattsson e Bosch (2000) explicam que os *frameworks* podem ser categorizados de formas distintas, de acordo com seu escopo, aplicação e forma de instanciação.

Fayad e Schmidt (1997) e Fayad et al. (1999) classificam os *frameworks* em três categorias, de acordo com seu escopo: *frameworks* de infra-estrutura de sistemas, *frameworks* de integração de *middleware* e *frameworks* de aplicações corporativas.

Frameworks de infra-estrutura de sistemas simplificam o desenvolvimento de sistemas portáveis e eficientes, tais como sistemas operacionais, sistemas de comunicação, ferramentas de interfaces de usuário e de processamento de linguagens. Normalmente, estes *frameworks* não são destinados ao cliente final, mas utilizados internamente em uma organização, durante o desenvolvimento de *software*.

Frameworks de integração de *middleware* são geralmente utilizados para integrar aplicações e componentes distribuídos. Estes *frameworks* trazem ao desenvolvedor facilidades que permitem a modularização e reutilização de *software*, de forma a se trabalhar em um ambiente distribuído.

Os *frameworks* de aplicações corporativas são utilizados em domínios mais extensos, como sistemas de telecomunicações, aviação, engenharia financeira, manufatura etc, constituindo o sistema base dessas aplicações. Comparados aos *frameworks* de infra-estrutura e de integração, *frameworks* de aplicações corporativas são complexos e possuem um alto custo de desenvolvimento.

Frameworks podem também ser classificados em horizontais e verticais, de acordo com a sua aplicação (FAYAD et al., 1999). *Frameworks* verticais são específicos de um domínio como, por exemplo, um sistema especialista, sendo, por isso, também chamados de *frameworks* especialistas. Já os *frameworks* horizontais não estão restritos a uma aplicação, podendo ser utilizados em diversos domínios como, por exemplo, o *framework* Hibernate (JBOSS, 2012), usado para persistência e mapeamento objeto-relacional em Banco de Dados.

De acordo com a maneira em que os *frameworks* são instanciados e/ou estendidos, eles podem, ainda, ser classificados em *frameworks* caixa-branca (*white-box*), caixa-preta (*black-box*) e caixa-cinza (*gray-box*). Nos *frameworks* caixa-branca, as aplicações usuárias estendem as classes do *framework* através de herança ou implementação de interfaces. Nos *frameworks* caixa-preta, as aplicações usuárias acessam o *framework* através de um mecanismo de configuração, como um arquivo XML. Os *frameworks* caixa-cinza combinam ambas as técnicas anteriores, sendo que uma parte da aplicação usuária utiliza um mecanismo de configuração, e outra parte faz uso de herança (FAYAD; SCHMIDT, 1997; MATTSSON; BOSCH, 2000).

Ainda de acordo com Fayad e Schmidt (1997), os principais benefícios trazidos pelos *frameworks* são:

- Modularidade: *frameworks* melhoraram a modularidade do sistema, encapsulando detalhes de implementação nas interfaces. Além disso, melhoram a qualidade do *software* desenvolvido por permitir a rápida localização dos pontos impactados por alterações de projeto e implementação.
- Reusabilidade: interfaces fornecidas pelos *frameworks* aumentam a reusabilidade por definirem componentes genéricos que podem ser reaproveitados para criar novas aplicações. A reutilização de componentes de *frameworks* pode render melhorias substanciais na produtividade do programador, bem como melhorar a qualidade, desempenho, confiabilidade e interoperabilidade de *software*.
- Extensibilidade: *frameworks* alcançam a extensibilidade fornecendo explicitamente métodos (*hook methods*) que permitem às aplicações estenderem suas interfaces. Estes métodos desacoplam as interfaces do *framework* do comportamento da aplicação em contextos específicos. A extensibilidade garante a personalização de novas funcionalidades em uma aplicação.
- Inversão de controle: os métodos definidos pelo usuário para personalização do *framework* geralmente são invocados de dentro do próprio *framework*, ao invés de serem chamados explicitamente pelo código do usuário. Dessa forma, o *framework* faz o papel de programa principal, coordenando a sequência de ações a serem realizadas pela aplicação.

Gamma et al. (1995) levantam as relações entre abordagens de desenvolvimento de *software* que objetivam a reutilização de componentes, como aplicações orientadas a objetos, *frameworks*, bibliotecas de classes e padrões de projeto.

Uma aplicação orientada a objetos delineia um *software* completo e executável, que atende a requisitos levantados nos documentos de especificação. Em contrapartida, *frameworks* são incompletos e não executáveis, fornecendo operações comuns a diversas aplicações, mas deixando pontos que devem ser estendidos pelas mesmas ao instanciar o *framework*.

Bibliotecas de classes (*libraries*) são um conjunto de classes relacionadas e reutilizáveis, projetadas para fornecer uma funcionalidade útil e de finalidade geral. Por outro lado, *frameworks* estão sempre relacionados a um domínio ou a algum aspecto de infra-estrutura ou *middleware*. Além disso, as classes de uma biblioteca são reutilizadas individualmente e de forma passiva, enquanto que as classes de um *framework* são utilizadas em conjunto, podendo ter papel ativo, assumindo o controle de execução de uma aplicação.

Padrões de projeto são elementos abstratos que descrevem a solução para um problema recorrente, explicando a intenção, custos e benefícios de uma abordagem. Enquanto que os *frameworks* podem ser materializados em código, somente exemplos da utilização de padrões de projeto podem ser codificados. Por conseguinte, padrões de projeto são elementos de arquitetura menores que *frameworks*. Um *framework* pode conter vários padrões de projeto, mas o inverso não é verdadeiro. *Frameworks* são altamente especializados, sempre possuindo um domínio de aplicação, ao passo que os padrões de projetos, ainda que possam vir a ser suficientemente específicos, não ditam a arquitetura de uma aplicação.

3.2 Padrões de projeto

As ideias de padrões de *software* vieram originalmente do campo da arquitetura. Christopher Alexander, um arquiteto austríaco, escreveu dois livros revolucionários que descrevem os padrões de arquitetura e planejamento urbano: *A Pattern Language: Towns, Buildings, Construction* (*Oxford University Press*, 1977) e *The Timeless Way of Building* (*Oxford University Press*, 1979). Os conceitos apresentados nesses livros são aplicáveis a vários campos fora da arquitetura, incluindo desenvolvimento de *software* (METSKER; WAKE, 2008).

Posteriormente, o trabalho de Beck e Cunningham (1987), inspirado pelas ideias de Alexander, introduziu cinco padrões para projeto de interface de usuário.

Já na década de 1990, Erich Gamma, Richard Helm, John Vlissides e Ralph Johnson começaram a trabalhar no que seria um dos livros de padrões de *software* mais referenciados de todos os tempos: *Design Patterns*. Publicado em 1994 e, muitas vezes chamado de *Gang of Four (GoF) Book*, este trabalho foi a maior influência individual sobre o assunto e popularizou a ideia de padrões de projeto (GRAND, 2002).

Assim, como definido por Gamma et al. (1995), *padrões de projeto* são soluções reutilizáveis para problemas recorrentes, que podem ser encontrados durante o desenvolvimento do *software*. Padrões de projeto são ideias que mostram como alcançar um objetivo, podendo ser aplicadas a determinados casos.

A seguir, é apresentada uma breve descrição dos padrões de projeto utilizados no desenvolvimento do *framework* proposto no Capítulo 4.

3.2.1 *Bridge*

O padrão *Bridge* é focado na modelagem de uma abstração. A palavra 'abstração' refere-se a(s) classe(s) que depende(m) de um conjunto de operações abstratas, onde diversas implementações deste conjunto de operações são possíveis.

A forma mais comum de implementar uma abstração é criar uma hierarquia, em que uma classe abstrata, no topo, define as operações e cada subclasse provê sua distinta implementação deste conjunto de operações. Entretanto, quando é necessário estender a hierarquia criada, esta abordagem torna-se insuficiente ou problemática.

Neste caso, pode-se criar uma ponte (*Bridge*), movendo algumas operações do conjunto para uma interface, de forma que as abstrações dependam da implementação da interface (METSKER; WAKE, 2008).

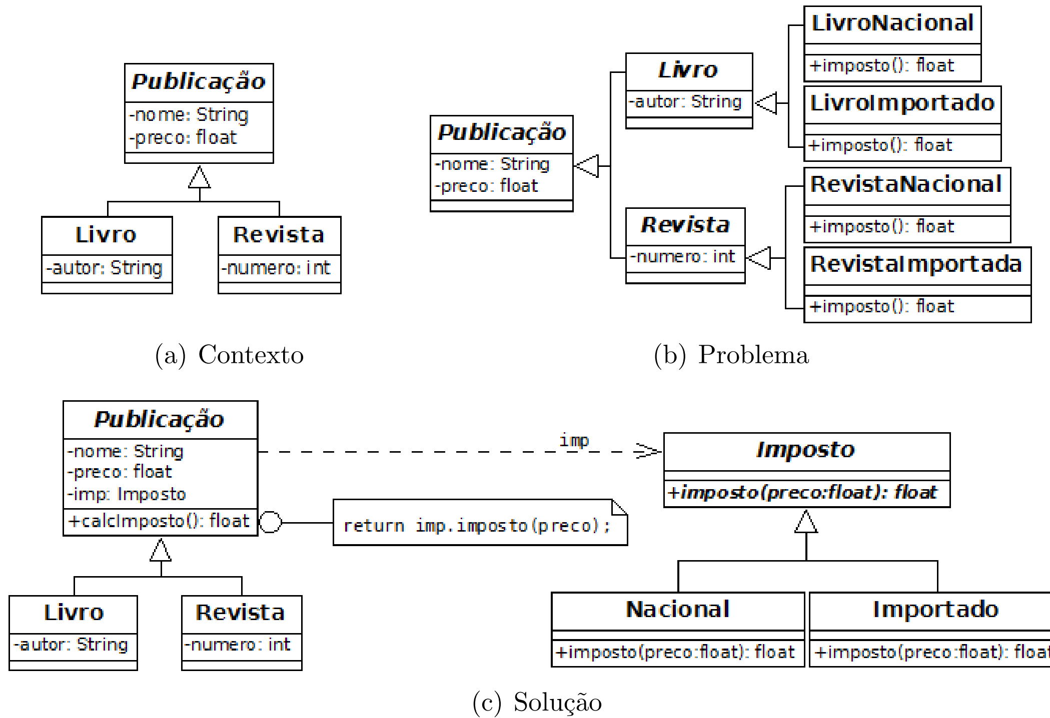
A intenção do padrão *Bridge* é desacoplar uma abstração da implementação de seu conjunto de operações, de forma que a abstração e suas implementações possam variar de forma independente (GAMMA et al., 1995). A seguir, é dado um exemplo de aplicação deste padrão.

Imagine um contexto onde as classes do modelo sejam Livros e Revistas, como ilustrado pela Figura 3.1(a). Suponha, agora, que seja necessário calcular o imposto sobre a venda desses Livros e Revistas. Considere, ainda, que taxas de imposto são aplicadas diferenciadamente para Publicações Nacionais e Importadas. Uma extensão de classes, como na Figura 3.1(b), torna-se problemática por duas razões:

1. Para suportar o cálculo dos impostos, deve-se implementar *duas* novas classes, uma para a publicação Nacional e outra para a publicação Importada. Mais ainda, cada novo tipo de Publicação que vier a ser futuramente adicionada ao sistema (por exemplo, Jornal), necessita da criação de duas classes adicionais (Jornal Nacional e Jornal Importado), totalizando três novas classes.

2. Esta hierarquia (Figura 3.1(b)) torna o código do usuário dependente do modelo de classes. Sempre que um cliente criar uma Publicação, instancia uma classe concreta, que tem uma implementação específica do cálculo do imposto. Além disso, o cliente necessita de ter o conhecimento de qual classe concreta deve ser instanciada.

O padrão *Bridge* resolve esses problemas colocando a abstração Publicação e as implementações de Impostos em hierarquias de classes separadas, como modelado na Figura 3.1(c). Todos os cálculos de impostos das Publicações são implementados em termos das operações da interface Imposto. Isso desacopla as abstrações de Publicação das diferentes taxas de Imposto. O relacionamento entre as classes Publicação e Imposto chama-se *Bridge*, pois ele forma uma ponte entre as duas hierarquias de classes, permitindo que variem independentemente.



(a) Contexto (b) Problema

(c) Solução

Figura 3.1: Exemplo do padrão *Bridge*
Fonte: O autor

O padrão *Bridge* tem as seguintes consequências (GAMMA et al., 1995):

1. Desacopla a interface da implementação: uma implementação não fica permanente presa a uma interface. O desacoplamento da abstração e implementação elimina dependências em tempo de compilação. Mudar uma classe de implementação não requer que as classes de abstração ou seus clientes sejam recompilados.

2. Melhora a extensibilidade: as hierarquias de abstração e implementação podem ser estendidas independentemente.

3. Oculta detalhes de implementação: as particularidades operacionais do sistema são isoladas nas classes de implementação, sendo transparentes aos clientes das classes de abstração.

3.2.2 *Decorator*

A fim de estender um determinado código, muitas vezes são adicionadas novas classes ou métodos. Entretanto, pode ser necessário adicionar a um objeto um novo comportamento, em tempo de execução. Alguns padrões de projeto, como o *Interpreter* (GAMMA et al., 1995), permite compor um novo objeto executável cujo comportamento muda radicalmente, dependendo do modo em que é feita essa composição. Em outros casos, as mudanças a serem

adicionadas a um objeto são sutis, causando pequenas variações em seu comportamento. O padrão *Decorator* atende a essa necessidade (METSKER; WAKE, 2008).

A intenção do padrão *Decorator* é permitir adicionar responsabilidades a objetos dinamicamente, em tempo de execução (GAMMA et al., 1995). A seguir, é apresentado um exemplo de uso deste padrão (FREEMAN et al., 2004).

Considere um empresa especializada na venda de cafés, em que seu modelo de classes foi inicialmente projetado conforme a Figura 3.2.

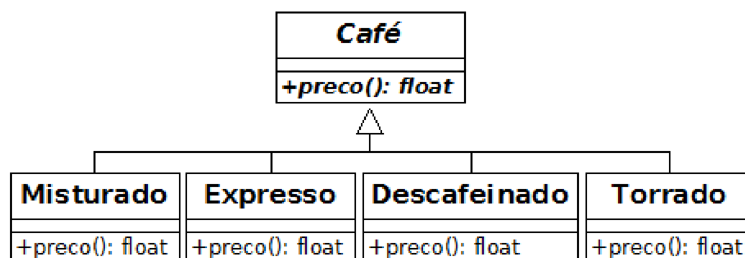


Figura 3.2: Contexto de aplicação do padrão *Decorator*
Fonte: Adaptada de Freeman et al. (2004)

Devido a qualidade de seus produtos, a empresa vivenciou um súbito crescimento e passou a oferecer a seus clientes algumas opções de "combinação", como chocolate, creme, leite, espuma etc. Visto que a empresa cobra um valor adicional por cada combinação, o modelo inicial de classes foi estendido conforme a Figura 3.3.

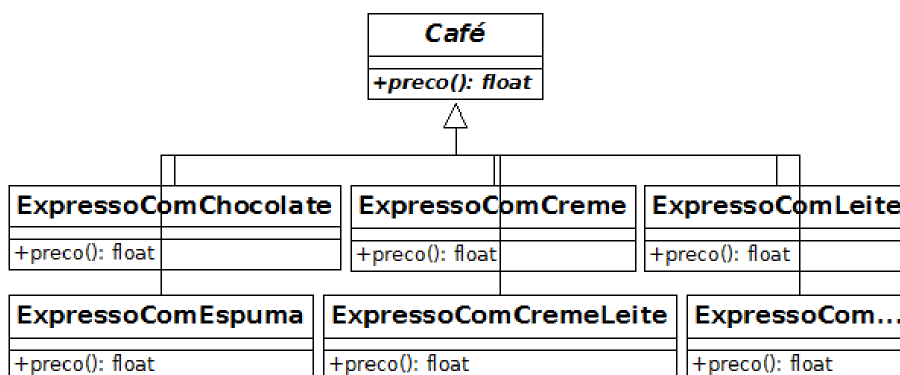


Figura 3.3: Extensão do modelo de classes da Figura 3.2
Fonte: Adaptada de Freeman et al. (2004)

A Figura 3.3 mostra somente *algumas* possíveis combinações com o tipo Expresso. Quando os tipos de cafés Descafeinado, Torrado e Misturado forem adicionados, com suas respectivas combinações, haverá uma explosão de classes. Esta modelagem, além de problemática, ainda levanta algumas outras questões como: o que acontece quando o preço do leite subir? O que acontece quando uma nova combinação for criada? O que acontece se o cliente pede um Expresso + Creme duplo?

A aplicação do padrão *Decorator* resolve o problema do modelo de classes da empresa de cafés, como ilustrado pela Figura 3.4.

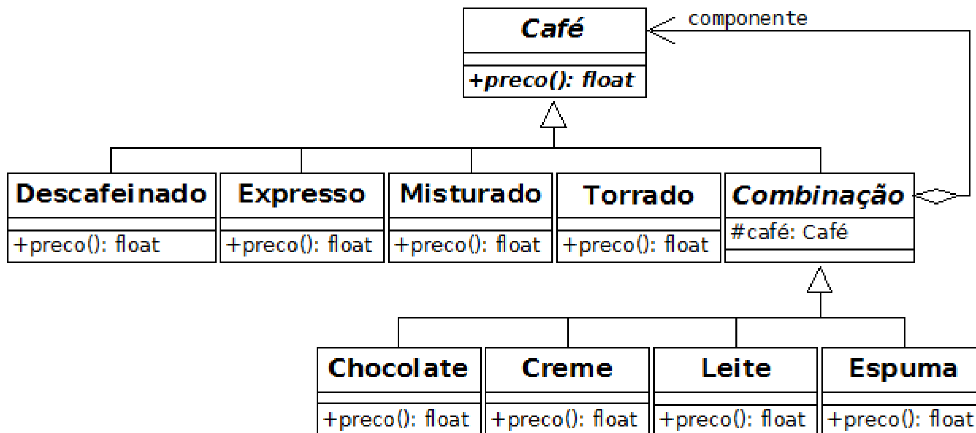


Figura 3.4: Exemplo do padrão *Decorator*
Fonte: Adaptada de Freeman et al. (2004)

Na Figura 3.4, a classe Café é chamada de 'componente'. Um 'componente' pode ser usado sozinho ou englobado por um 'decorador'. Cada tipo de café (Expresso, Torrado etc) é um 'componente concreto' e herda da classe abstrata Café, implementando o método 'preço'. Aos 'componentes concretos' é que serão adicionados novas responsabilidades ou comportamentos. A classe Combinação é chamada de 'decorador', espelhando o objeto que está decorando (Café). Além disso, os 'decoradores' também implementam o método 'preço', podendo, por polimorfismo, tratar qualquer bebida que herde da classe Café. Por fim, as subclasses de Combinação são os 'decoradores concretos' e mantêm uma variável de instância para cada 'componente' decorado.

Através da aplicação do padrão *Decorator* (Figura 3.4), as questões anteriores podem ser facilmente respondidas:

- *O que acontece se o preço do leite subir?* Com o uso do *Decorator*, basta alterar apenas uma única classe (Leite) para que passe a devolver o novo preço. No modelo de classes da Figura 3.3, dezenas de classes, que incluem leite como combinação, deveriam ser alteradas.

- *E se uma nova combinação for criada?* Com o uso do *Decorator*, a nova combinação (por exemplo, Caramelo), herda da classe abstrata Combinação (Figura 3.4) e implementa as operações necessárias (no caso, o método 'preço'). O mesmo vale para novos tipos de café (por exemplo, Colombiano), que devem estender a classe base Café.

- *E se o cliente pede um Expresso + Creme duplo?* Segue-se o padrão *Decorator*. Em termos de codificação, seria algo do tipo: `Cafe c = new Expresso(); c = new`

`Creme(c); c = new Creme(c);`, onde, cria-se um tipo de café (Expresso) e adiciona-se duas vezes a combinação Creme. Cada combinação adiciona o seu preço ao preço antigo do componente.

As consequências do padrão *Decorator* são (GAMMA et al., 1995):

1. Maior flexibilidade do que herança estática: o padrão *Decorator* fornece uma maneira mais flexível de acrescentar novas responsabilidades a objetos do que pode ser feito com herança estática (múltipla). O uso de herança requer uma nova classe base para cada responsabilidade adicional, dando origem a muitas classes, aumentando a complexidade do sistema.

2. Evita classes sobrecarregadas de características na parte superior da hierarquia: as classes de mais alto nível na hierarquia são mantidas simples, não precisando definir diversos atributos para representar todas as possíveis responsabilidades necessárias à aplicação.

3. Um decorador e seu componente não são idênticos: um decorador funciona como um envoltório transparente. Porém, do ponto de vista de identidade de um objeto, um componente decorado não é idêntico ao próprio componente.

4. Grande quantidade de pequenos objetos: o uso de *Decorator* resulta em um sistema composto por uma grande quantidade de pequenos objetos parecidos. Esses objetos diferem apenas na maneira como são interconectados, e não nas suas classes ou no valor de suas variáveis.

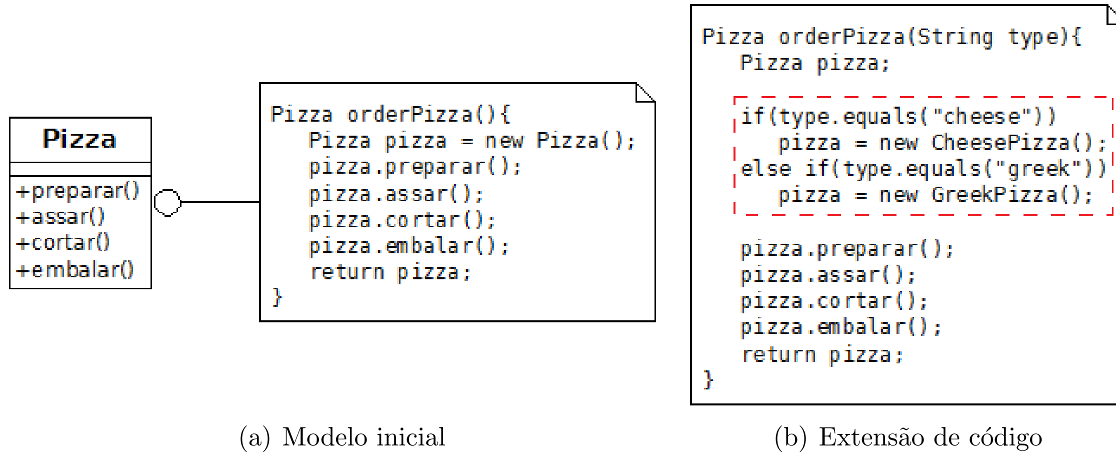
3.2.3 *Factory Method*

Quando uma classe é projetada, normalmente, métodos construtores permitem ao cliente instanciá-la. Entretanto, algumas vezes, o cliente necessita de um determinado objeto, mas não sabe, ou não deveria saber, quais das muitas classes disponíveis em uma aplicação deve ser instanciada. Este impasse é resolvido pelo padrão *Factory Method* (METSKER; WAKE, 2008).

O padrão *Factory Method* permite definir uma interface para criação de objetos, mas transfere a responsabilidade sobre quais instâncias devem ser criadas para as subclasses desta interface (GAMMA et al., 1995). Um exemplo de aplicação deste padrão é mostrado a seguir (FREEMAN et al., 2004).

Considere uma pizzaria em que a classe modelo e o trecho de código responsável pelo pedido de pizzas seja como o da Figura 3.5(a). Apesar de consistente, o método descrito na Figura 3.5(a) permite que apenas pizzas "genéricas" sejam produzidas. Como a pizzaria

vende pizzas de diversos sabores, deve-se adicionar ao método da Figura 3.5(a) um trecho de código que *determina* o tipo apropriado de pizza e depois *cria* a pizza, como mostrado na Figura 3.5(b) (escrito em Java).



(a) Modelo inicial (b) Extensão de código

Figura 3.5: Contexto de aplicação do padrão *Factory Method*
Fonte: Adaptada de Freeman et al. (2004)

Na Figura 3.5(b), as instruções condicionais, com base num tipo de Pizza, instanciam uma classe concreta que será atribuída à variável de instância de Pizza. Dessa forma, sempre que a pizzaria passar ou deixar de oferecer um certo tipo de pizza, o trecho quadriculado da Figura 3.5(b) deverá ser alterado. Analisando o método 'orderPizza', percebe-se que existe um trecho de código que será sempre fixo, o que lida com a fabricação das pizzas, e outro trecho que sofrerá variações temporárias, o que lida com instanciação das pizzas. Claramente, lidar com qual classe concreta deve ser instanciada é bagunçar o método 'orderPizza' e impedir que ele seja reutilizável e fechado para modificações.

O uso do padrão *Factory Method* organiza o modelo de classes da pizzaria, como mostrado na Figura 3.6. A classe FactoryPizza passa a receber a parte variável do método 'orderPizza', da Figura 3.5(b). Por consequência, todas as requisições dos clientes por classes concretas ficam encapsuladas no método 'createPizza' da classe FactoryPizza, evitando que alterações sejam realizadas em outras classes. O método 'createPizza' é chamado de *Factory Method*.

À primeira vista, pode parecer que a solução apresentada pela Figura 3.6 apenas "empurra" o problema da modificação de código para outro objeto: a classe FactoryPizza. Entretanto, a mesma figura mostra somente o método 'orderPizza' como cliente da fábrica. Com a expansão da aplicação, pode vir a existir uma classe DescriptionPizza acessando a fábrica para obter a descrição de cada pizza, uma classe MenuPizza que precise obter o preço de cada pizza, uma classe DeliveryPizza, responsável por gerenciar as entregas etc. Todas as classes futuras seriam clientes da classe fábrica. Assim, ao encapsular a criação de pizzas em uma classe, haverá apenas um único local a ser alterado quando a aplicação sofrer modificações.

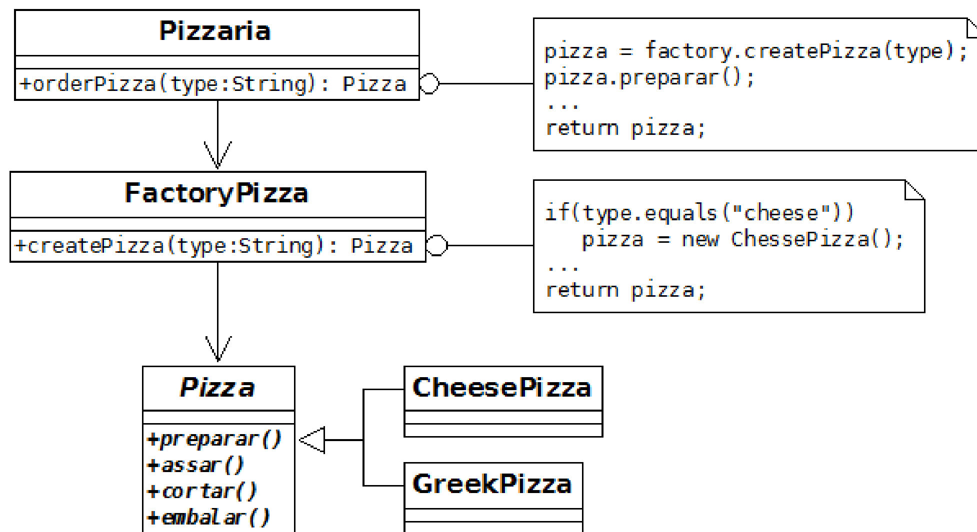


Figura 3.6: Exemplo do padrão *Factory Method*
Fonte: Adaptada de Freeman et al. (2004)

O padrão *Factory Method* tem as seguintes consequências (GRAND, 2002):

1. Independência na instanciação: a classe que requisita a criação de um objeto é independente da classe concreta do objeto produto.

2. O conjunto de classes produto que podem ser instanciadas pode ser alterado dinamicamente.

3. A flexibilidade entre o início da criação do objeto e a determinação de qual classe instanciar torna mais fácil dar manutenção na aplicação.

3.2.4 *Template Method*

Métodos comuns têm corpos que definem uma sequência de instruções. Também é bastante comum para um método invocar métodos no objeto atual e em outros objetos. Neste sentido, os métodos comuns são "*templates*" que descrevem uma série de instruções a serem seguidas. O padrão *Template Method*, no entanto, introduz uma forma mais específica de modelar os métodos comuns.

Ao escrever um método, pode ser de interesse do programador definir a estrutura, o esqueleto deste método, mas deixar certas etapas "em aberto", permitindo que haja diferenças nas formas em que esses passos sejam implementados. Esses passos não definidos podem ser implementados ou invocados a partir de classes abstratas ou interfaces, definidas pelo usuário. Essa abordagem produz um "*template*" mais rígido, que define especificamente quais os passos de um algoritmo outras classes podem ou devem fornecer (METSKER; WAKE, 2008).

A intenção de um *Template Method* é definir o esqueleto de um algoritmo, postergando alguns passos para as subclasses. Dessa forma, subclasses podem redefinir certos passos de uma rotina, sem alterar a estrutura da mesma (GAMMA et al., 1995).

Na Figura 3.7, temos um exemplo de uso de um *Template Method*. Na classe Confeito, o método 'lucro' calcula a diferença entre o preço de venda e o custo do mesmo. Dessa forma, 'lucro' define a estrutura, o esqueleto deste cálculo, deixando as implementações específicas dos métodos 'custo' e 'venda' para as classes concretas. Por esta razão, o método 'lucro' é um *Template Method* e os métodos 'custo' e 'venda' são conhecidos como 'operações primitivas' do *template*.

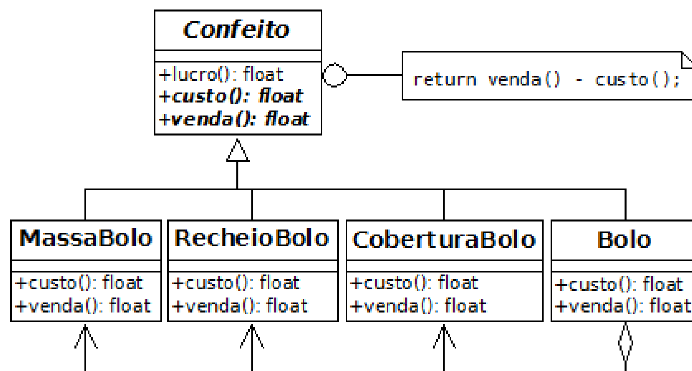


Figura 3.7: Exemplo do padrão *Template Method*
Fonte: O autor

As consequências do padrão *Template Method* são (GAMMA et al., 1995):

1. Inversão de controle: ao invés das subclasses invocarem métodos presentes na superclasse, a classe mãe é quem chama os métodos definidos pelas classes filhas.

2. Reuso: métodos *template* são importantes em bibliotecas de classes porque são meios para fatoração de componentes e comportamentos comuns.

3.2.5 *Curiously Recurring Template Pattern*

Batizado por Coplien (1995) e inicialmente observado na linguagem C++, o CRTP, acrônimo de *Curiously Recurring Template Pattern*, é mais uma técnica ou estilo de programação (idioma) que um padrão no sentido estrito da palavra. Este idioma combina o uso de herança com parâmetros *template* para gerar tipos flexíveis em tempo de compilação.

A intenção do CRTP é criar uma subclasse que herde da instanciação de uma classe base *template* usando a própria subclasse como argumento do *template*. Dessa forma, pode-se, em tempo de compilação, especializar a classe base usando a subclasse.

A Figura 3.8 mostra um exemplo de modelagem UML do uso de CRTP. ClasseBase é uma classe *template* cujo parâmetro é T. ClasseBase declara um método chamado *isSame*, responsável por verificar se duas referências são a mesma instância de um objeto. Por sua vez, Subclasse deriva de ClasseBase e passa a si própria como argumento do *template* através do estereótipo << *bind* >>. Este argumento será usado por ClasseBase na assinatura do método *isSame*.

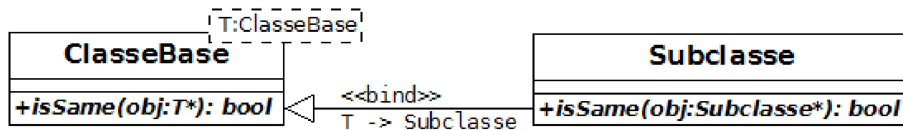


Figura 3.8: Modelagem do CRTP

Fonte: O autor

O Código 3.1 mostra a implementação correspondente à Figura 3.8 em C++. A linha 7 invoca uma função especial da biblioteca Boost (BOOST, 2012) para verificar se o tipo T é derivado de ClasseBase. Na implementação de Subclasse, devido ao CRTP, a assinatura do parâmetro do método *isSame* foi modificada, recebendo um ponteiro de Subclasse no lugar do argumento T de ClasseBase.

Código 3.1: CRTP em C++

```
1  template<typename T>
2  class ClasseBase
3  {
4          public:
5                  ClasseBase()
6                  {
7                          BOOST_MPL_ASSERT(is_base_of<ClasseBase<T>, T>);
8                  }
9
10                 virtual bool isSame(T* obj) = 0;
11 };
12
13 class Subclasse : public ClasseBase<Subclasse>
14 {
15         public:
16                 bool isSame(Subclasse* obj)
17                 {
18                         return (this == obj);
19                 }
20 };
```

3.3 Considerações finais

Este Capítulo apresentou o conceito de *frameworks*. *Frameworks* tem como principal característica uma arquitetura flexível, semi-pronta, que pode ser personalizada pelas aplicações conforme suas necessidades. A customização de um *framework* é possível através da implementação e/ou extensão de seus pontos flexíveis (*hot spots*).

O uso de padrões de projeto possibilita a captura de conceitos gerais às aplicações, aumentando a qualidade e reduzindo o esforço de desenvolvimento do software. Além disso, os padrões de projeto permitem o projeto de um software reutilizável, uma vez que capturam o conhecimento de um determinado domínio, ajudando, inclusive, no desenvolvimento de *frameworks*.

Capítulo 4

O Framework Object-Injection

> Do you pine for the nice days of minix-1.1, when men were men and wrote their own device drivers? Are you without a nice project and just dying to cut your teeth on a OS you can try to modify for your needs? Are you finding it frustrating when everything works on minix? No more all-nighters to get a nifty program working? Then this post might be just for you.
>
> Linus Torvalds

Este Capítulo apresenta com detalhamento a principal contribuição deste trabalho, que é o *framework* Object-Injection, e explana os conceitos que guiaram a sua modelagem e desenvolvimento. Esta descrição não se atém a uma linguagem de programação particular, mas utiliza apenas os fundamentos da programação orientada a objetos. Objetivando solucionar questões semânticas e de encapsulamento, a maior parte do modelo de classes usa *Curiously Recurring Template Pattern* (CRTP) (Seção 3.2.5) e padrões de projeto. Os padrões de projeto, quando mencionados, comportam-se e referem-se aos descritos na Seção 3.2 e em Gamma et al. (1995).

A Seção 4.1 explana aspectos teóricos considerados durante a implementação. A Seção 4.2 apresenta a arquitetura do *framework* desenvolvido, que é formada por quatro módulos: Metaclasses, Armazenamento, Blocos e Dispositivos. Dando sequência, as Seções 4.3, 4.4, 4.5 e 4.6 apresentam o detalhamento desses módulos. Finalizando, a Seção 4.7 apresenta trabalhos similares a este e descreve suas principais características.

4.1 Conceitos iniciais

Esta Seção descreve os conceitos de índices primários, índices secundários e identificadores únicos universais, que foram empregados no desenvolvimento do *framework* proposto.

4.1.1 Índices primários e secundários

De acordo com Ramakrishnan e Gehrke (1999) e Garcia-Molina et al. (2008), um *índice primário* sobre um conjunto de campos determina a localização de um registro em um arquivo de dados. Por consequência, todos os índices que não são índices primários são chamados de *índices secundários*.

Seguindo as definições de Ramakrishnan e Gehrke (1999), neste trabalho, os índices primários estão associados à instância completa de um objeto, não havendo separação entre seu Identificador Único Universal (Seção 4.1.2) e seus atributos. De forma análoga, aos índices secundários associam-se todas as referências ou ponteiros para uma entrada de índice primário.

4.1.2 Identificador Único Universal

Um Identificador Único Universal (*Universally Unique Identifier* - UUID) é uma sequência de 16 *bytes*, ou 128 *bits*, sendo que cada *byte* pode ser codificado como um número inteiro não sinalizado. Esta sequência é representada por um conjunto de 32 dígitos hexadecimais divididos em cinco grupos separados por hifens, como 01234567-89AB-CDEF-0123-456789ABCDEF.

UUIDs foram originalmente apresentados no trabalho de Zahn et al. (1990) e posteriormente em OpenGroup (1994). Atualmente, UUIDs encontram-se especificados pela norma ISO 11578 (ISO/IEC, 1996) e foram recentemente estendidos pela ISO 9834-8 (ISO/IEC, 2008). Embora UUIDs sejam também chamados de GUIDs (*Globally Unique Identifier*), este termo não é usado nas normas e trabalhos que o definem.

Um UUID pode ser usado para várias finalidades, desde a marcação de objetos com uma vida útil extremamente curta, até a identificação confiável de objetos persistentes através de uma rede. Nesses casos, um UUID pode ser parte de ou ser o próprio valor de identificador de objeto (OID) ou usado em um *Uniform Resource Name* (URN) (LEACH et al., 2005).

Como descrito em ITU (2004), três algoritmos são especificados para a geração de UUIDs únicos, utilizando diferentes mecanismos para garantir essa unicidade. Estes algoritmos produzem diferentes versões de um UUID.

O primeiro e mais comum algoritmo produz a assim chamada versão baseada no tempo. A geração de UUIDs e garantia de exclusividade é baseada no *Universal Time Coordinated* (UTC) em um mesmo computador. Em computadores diferentes, a geração baseia-se também no valor do MAC (*Media Access Control*) *addresses* do *hardware* de rede.

O segundo mecanismo é uma versão baseada em texto. Este algoritmo utiliza técnicas de *hash* criptográfico para produzir o valor de 128 *bits* a partir de um nome (texto) que não se repita em um contexto.

O terceiro e mais simples algoritmo é baseado em geradores de números aleatórios para produzir os valores que formam os 128 *bits* de um UUID.

ITU (2004) ainda afirma que, em conformidade com a descrição destes algoritmos, se forem gerados 100 trilhões de UUIDs a cada nanosegundo, seriam necessários pouco mais de 3,1 trilhões de anos para esgotar todas as possibilidades.

UUIDs gerados pelo primeiro algoritmo (baseado em tempo) são compostos por uma sequência de seis campos ordenados e concatenados. Esses campos são:

- *TimeLow*: formado pelos 32 *bits* mais significativos, este campo armazena os *bits* de mais baixa ordem do valor temporal, dado em UTC.
- *TimeMid*: são os 16 *bits* que definem os *bits* centrais do valor temporal, dado em UTC.
- *VersionAndTimeHigh*: este campo, composto por 16 *bits*, armazena a versão do UUID (4 *bits*), seguida pelos *bits* de mais alta ordem do valor temporal, dado em UTC (12 *bits*).
- *VariantAndClockSeqHigh*: estes 8 *bits* armazenam um valor de variação (2 *bits*), seguido pelos *bits* de mais alta ordem da sequência do *clock* do processador (6 *bits*), no momento da geração do UUID.
- *ClockSeqLow*: com tamanho de 8 *bits*, este campo armazena os *bits* de mais baixa ordem da sequência do *clock* do processador, no momento da geração do UUID.
- *Node*: os 48 *bits* menos significativos armazenam informações sobre o MAC *address* do *hardware* de rede.

No *framework* Object-Injection, UUIDs são usados para definir a chave de indexação de um objeto e o valor de identificador de objeto (OID), como detalhado na Seção 4.3.

4.2 Arquitetura modularizada

A arquitetura do *framework* Object-Injection é baseada nos conceitos de índices primários e secundários. Neste trabalho, os índices primários são responsáveis pelo armazenamento das entidades persistentes, enquanto que os índices secundários são responsáveis pelo armazenamento das chaves indexadas.

Os índices primários armazenam os valores dos atributos de um objeto e são indexados através de um UUID. O índice primário de uma classe é determinado pela implementação da interface *Entity* (Seção 4.3). Neste *framework*, a principal estrutura de índices primários é um *Hash* Extensível (Seção 2.1.2), que usa como chave o valor do UUID do objeto.

Os índices secundários replicam apenas os atributos dos objetos que definem uma chave de indexação. Essa chave compõe a hierarquia das estruturas de indexação, normalmente baseadas em árvores e definidas por nós internos e nós folha. Os nós internos contêm as chaves de roteamento, ao passo que os nós folha mantêm todos os valores de chaves com os seus respectivos UUIDs dos objetos. Assim, a partir de uma chave armazenada em um índice secundário, é possível obter o seu UUID que, por sua vez, possibilita recuperar a instância do objeto procurado em um índice primário.

O índice secundário de uma classe é definido pela implementação das sub-interfaces de *Key* (Seção 4.3). Esses índices manipulam chaves que tem relação de ordem, relação métrica e relação espacial. As chaves que apresentam relação de ordem constituem uma Árvore B. A Árvore M é formada por chaves que tem uma relação métrica. Por fim, as chaves que tem relação espacial fazem parte da Árvore R, podendo, ainda, compor uma Árvore M.

O *framework* Object-Injection encontra-se particionado em quatro módulos de abstração, organizados de acordo com a Figura 4.1. O módulo das Metaclasses (Seção 4.3) define as entidades persistentes e o domínio das chaves indexadas. O módulo de Armazenamento (Seção 4.4) define estruturas de índices primários para as entidades persistentes e estruturas de índices secundários para as chaves indexadas. Estas estruturas de indexação são responsáveis por gerenciar e organizar os dados do módulo das Metaclasses, que são mantidos no módulo de Blocos. O módulo de Blocos (Seção 4.5) define a maneira pela qual as entidades persistentes e chaves indexadas são armazenadas em blocos ou páginas. Por fim, o módulo de Dispositivos (Seção 4.6) define os recursos computacionais responsáveis pelo armazenamento físico das estruturas de índices. Além disso, este módulo fabrica blocos requisitados pelo e repassados para o módulo de Armazenamento.

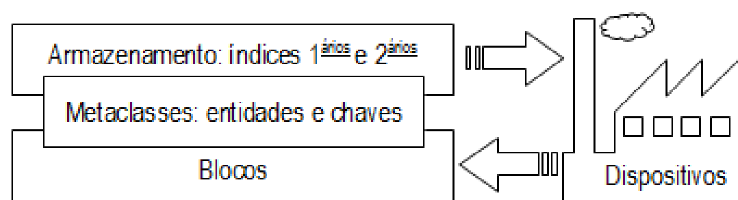


Figura 4.1: Organização dos módulos

Fonte: O autor

4.3 O módulo das Metaclasses

O módulo das Metaclasses define o vínculo entre as classes de aplicação do usuário e o *framework*, expresso através de entidades persistentes e chaves indexadas.

Todas as classes de aplicação do usuário devem ser especializadas de forma a implementar uma interface comum (*Entity*), tornando-se entidades persistentes. A Figura 4.2 ilustra o

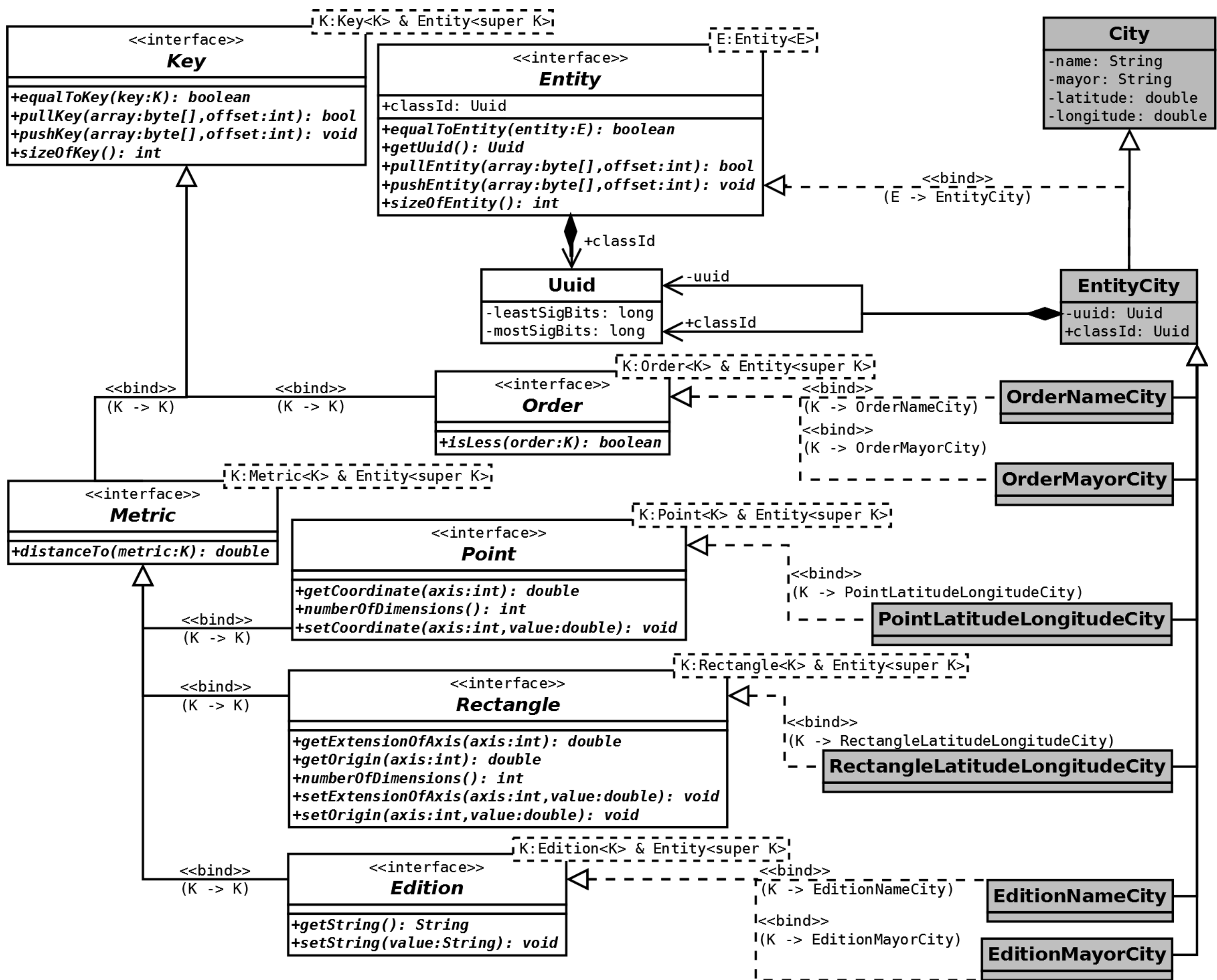


Figura 4.2: Hierarquia do módulo das Metaclasses

Fonte: O autor

cenário das entidades persistentes. A classe destacada *City* exemplifica e representa uma classe definida pelo usuário, isto é, aquilo que o usuário desejar tornar persistente. A fim de não realizar modificações nas classes definidas pelo usuário, é realizada uma herança múltipla através das mesmas. Na Figura 4.2, a classe *EntityCity* especializa a classe concreta *City* e implementa a interface *Entity*. Assim, *EntityCity* possui as mesmas características da classe *City* e implementa os métodos requeridos pela interface *Entity*. Como *EntityCity*, as classes do usuário seguem o padrão CRTP, isto é, usam elas próprias como argumento do *template*.

Sendo *Entity* uma classe parametrizada, o parâmetro genérico garante que somente objetos de uma mesma classe possam ser comparados com outros. Dessa forma, previne-se comparações entre objetos distintos, visto que o resultado seria falso, em todo caso. Para verificar se duas entidades persistentes são iguais e evitar armazenar duplicatas, todos os objetos implementam o método *equalToEntity*. A intenção deste método não é verificar se duas entidades *são* o mesmo objeto, mas apenas verificar se duas entidades *têm* os mesmos valores de atributos.

Cada classe derivada de *Entity* possui dois atributos do tipo UUID. O primeiro, chamado *uuid*, é usado pelas estruturas de dados (especializadas de *EntityStructure* (Seção 4.4)) para recuperar a instância de um objeto armazenado. O outro, nomeado *classId*, é um atributo de escopo de classe (*static*), cujo valor deve ser único entre todas as derivações e implementações da classe *Entity*, independente da aplicação. O *classId* é utilizado para garantir que um objeto recuperado é, de fato, uma instância da classe correta.

É permitido somente às entidades persistentes terem domínios de chave associados a seus atributos. Para definir um domínio de chave, uma entidade deve ser derivada em uma sub-interface de *Key*, como é requerido pelo CRTP. Os métodos da classe *Key* são idênticos aos da classe *Entity*, porém enquanto uma entidade (*Entity*) compara e serializa todo o conjunto de atributos da classe, uma chave (*Key*) realiza o mesmo procedimento, mas apenas com o atributo de indexação. A classe *Key* é especializada nas classes *Order* e *Metric*.

Chaves derivadas de *Order* indexam um conjunto de objetos $\mathcal{S}$ que satisfazem a relação de ordem total (Seção 2.1.1). As propriedades da relação de ordem total (transitividade, anti-simetria e totalidade) são alcançadas e implementadas através do método *isLess*, da interface *Order*. Este é um método booleano usado para comparar dois objetos, a fim de estabelecer uma ordenação total sobre o conjunto de objetos $\mathcal{S}$. Desta maneira, a implementação do método *isLess* determina qual(is) atributo(s) de um objeto será(ão) usado(s) como índice.

Embora na Figura 4.2 alguns métodos estejam omitidos por razões de espaçamento, a classe *OrderNameCity* é uma chave (*Key*) para um objeto *City*, onde o atributo de indexação é o nome (*name*) da cidade *City*. Em *OrderNameCity*, o método *isLess* recebe como parâmetro uma instância de outro objeto do tipo *OrderNameCity* e compara se seu atributo *name* vem lexicograficamente antes do atributo *name* do argumento *OrderNameCity*. De forma análoga, na classe *OrderMayorCity*, as cidades são indexadas pelo nome do prefeito, e assim por diante, de acordo com o atributo que o usuário define como índice. Estas classes, derivadas de *Order*, podem ser indexadas em uma estrutura de índices secundários como a Árvore B.

De forma similar, as chaves derivadas de *Metric* indexam um conjunto de objetos $\mathcal{D}$ onde não é possível estabelecer uma relação de ordem total, ou mesmo parcial. Contudo, esses objetos podem ser relacionados de acordo com uma medida de sua proximidade, similaridade ou dissimilaridade, dada por uma função de distância ou métrica. As métricas seguem as propriedades descritas na Seção 2.2 (não-negatividade, simetria, reflexividade, positividade e desigualdade triangular) e são implementadas pelo método *distanceTo*, da classe *Metric*.

Chaves métricas são especializadas em edição de caracteres, pontos e retângulos, respectivamente, na Figura 4.2, as classes *Edition*, *Point* e *Rectangle*. Enquanto que a classe *Edition* possui métodos para recuperar e atribuir valor a uma *string* de indexação, *Point* e *Rectangle* possuem métodos que devolvem suas dimensionalidades, coordenadas de origem e medidas de extensão ao longo do espaço.

Seguindo o exemplo da Figura 4.2, a classe *PointLatitudeLongitudeCity* representa uma chave (*Key*) de um objeto *City*, onde as cidades são definidas por um ponto (dado pelos valores de latitude e longitude), e o método *distanceTo* computa o quão distante uma cidade está da outra, de acordo com a distância Euclidiana. Na classe *RectangleLatitudeLongitudeCity*, o principal interesse é indexar cidades como regiões do espaço, usando sua latitude, longitude e dimensões, para fins de posicionamento e localização espacial. As classes *EditionNameCity* e *EditionMayorCity* indexam cidades seguindo uma distância entre *strings* (por exemplo, Levenshtein (1966) (Apêndice A)) de seus nomes e nomes dos prefeitos, respectivamente. As chaves *EditionNameCity*, *EditionMayorCity*, *PointLatitudeLongitudeCity* e *RectangleLatitudeLongitudeCity* são indexadas por uma Árvore M, sendo que as duas últimas também podem ser inseridas em uma Árvore R.

Entidades persistentes e chaves indexadas são armazenadas em índices primários e secundários usando suas formas serializadas. Os detalhes sobre este formato são discutidos na Seção 4.5. A serialização de objetos é realizada através de duas classes auxiliares que manipulam o fluxo de dados serializados. Ambas as classes derivam da classe abstrata *Stream*, que contém seus atributos em comum, como ilustrado pela Figura 4.3

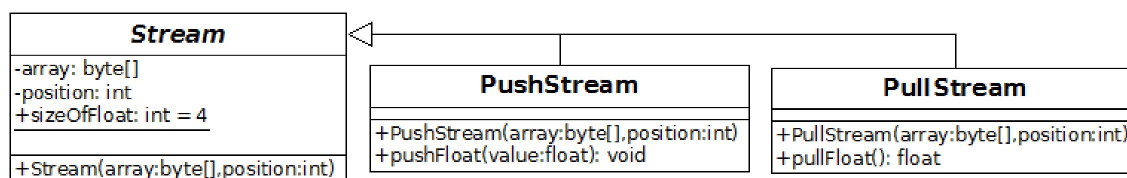


Figura 4.3: Hierarquia de serialização

Fonte: O autor

A classe *Stream*, no topo da hierarquia de serialização, recebe um vetor de *bytes* e uma posição inicial para efetuar operações neste vetor. Esta classe também é responsável por definir constantes numéricas para *cada* um dos tipos de dados disponíveis, representando seus tamanhos em *bytes* quando serializados no vetor. Como modelado na Figura 4.3, o vetor de *bytes* e a posição inicial são passados para a classe *Stream* através de seu construtor. A constante *sizeOfFloat* exemplifica o comprimento do tipo de dado `float` quando convertido e armazenado sob o formato de *bytes*. Constantes para os outros tipos, não apenas tipos primitivos, mas todos os tipos de dados usados por objetos, como UUIDs, são declaradas pela classe *Stream*.

As especializações da classe *Stream* fornecem as primitivas básicas para serializar e desserializar objetos. A classe *PushStream* é utilizada para serializar, colocar todos os valores de atributos de um objeto num vetor de *bytes*. Cada operação de serialização inicia a escrita na posição inicial recebida pelo construtor e incrementa essa posição para escritas futuras. Como mostrado na Figura 4.3, o método *pushFloat* recebe um valor do tipo `float` e o escreve no vetor, utilizando quatro *bytes* (*sizeOfFloat*).

Similarmente, a classe *PullStream* é utilizada para desserializar, extrair do vetor de *bytes* os valores de atributos de objetos. A cada operação de desserialização, a variável *position* também é incrementada, para permitir leituras futuras. Na Figura 4.3, o método *pullFloat* lê quatro *bytes* do vetor (*sizeOfFloat*), os converte para o tipo primitivo `float` e devolve este valor.

Todas as entidades persistentes e chaves indexadas implementam, respectivamente, os métodos *pullEntity*/*pushEntity* e *pullKey*/*pushKey*, que são responsáveis pela (des)serialização, usando as classes *PullStream* e *PushStream*.

Uma vez que o esquema de (des)serialização está definido, o objeto torna-se apto a realizar essas operações por si próprio. Desta forma, os objetos não necessitam de ser repassados para outras estruturas a fim de realizar a (des)serialização, evitando realizar cópias de objetos na memória e trazendo um ganho de desempenho.

As relações entre os objetos podem ser expressas de duas maneiras: usando uma única referência ou uma coleção de referências. Na primeira situação, um objeto retém a referência para algum outro objeto, podendo (des)serializá-lo diretamente, visto que os objetos são responsáveis por (des)serializar eles próprios. A segunda situação é definida através de vetores ou coleções de objetos. Classes com uma referência adicionam em sua serialização apenas o UUID do objeto referenciado. De forma análoga, classes com coleções de referências (des)serializam toda a coleção de UUIDs para estes objetos.

O mecanismo de (des)serialização também provê métodos *pull* e *push* para vetores de tipos primitivos e vetores de objetos, em qualquer dimensão, sendo estes vetores completos ou incompletos. Coleções de objetos também podem ser (des)serializadas de acordo com o seu conteúdo.

4.4 O módulo de Armazenamento

O módulo de Armazenamento especifica a maneira pela qual as estruturas de índices primários e secundários são implementadas.

Neste módulo, as estruturas de dados manipulam coleções de blocos, a fim de organizar e armazenar os objetos definidos na Seção 4.3. A especificação do módulo de Armazenamento é ilustrada na Figura 4.4.

A classe *Structure*, no topo da hierarquia, é dependente da classe *Workspace* (Seção 4.6), para criar, recuperar e atualizar blocos. Dessa forma, para instanciar uma *Structure*, é necessário que exista um *Workspace*. A classe *Structure* é especializada em *EntityStructure* e *KeyStructure*. *EntityStructure* é um índice primário, que manipula entidades persistentes, ao passo que *KeyStructure* é um índice secundário, gerenciando as chaves indexadas.

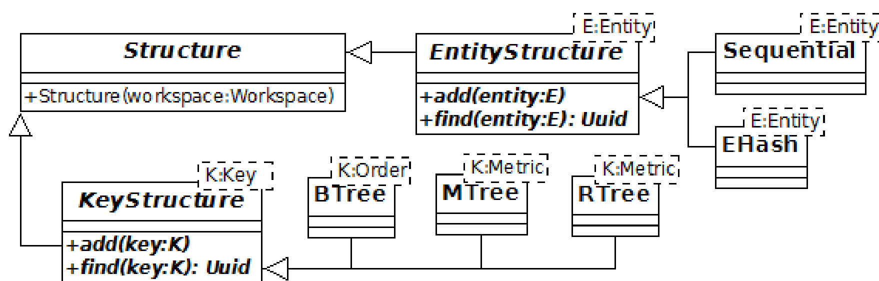


Figura 4.4: Hierarquia do módulo de Armazenamento
Fonte: O autor

Uma *EntityStructure* define as operações comuns (por exemplo, *add* e *find*) a todas as estruturas de dados responsáveis pelo armazenamento de um objeto completo (instância), tais como as classes *Sequential* e *EHash*. De maneira similar, a classe *KeyStructure* define as operações comuns para as estruturas de dados responsáveis pelo armazenamento de índices, tais como as classes *BTree*, *MTree* e *RTree*.

A classe *Sequential* implementa uma lista sequencial circular duplamente encadeada, enquanto que a classe *EHash* é um *Hash* Extensível (Seção 2.1.2). Ambas são utilizadas para armazenar entidades persistentes, embora o *Hash* Extensível tenha um desempenho superior ao da Lista Sequencial. As classes *BTree*, *MTree* e *RTree* são estruturas de índices secundários usadas para armazenar chaves indexadas. Elas implementam, respectivamente, a Árvore B+ (Seção 2.1.1), Árvore M (Seção 2.2.1) e Árvore R (Seção 2.1.3).

Esta hierarquia permite que outras estruturas de dados sejam facilmente criadas e/ou implementadas, simplesmente através de uma especialização das classes *EntityStructure* ou *KeyStructure* e implementando os métodos requeridos pelas mesmas. Além disso, esta modelagem permite que várias estruturas de dados compartilhem o mesmo *Workspace*, gerenciando entidades e chaves.

4.5 O módulo de Blocos

Sendo um *bloco* a unidade básica de transferência de dados entre os dispositivos de armazenamento e o *framework*, este módulo define (i) a estrutura do bloco, (ii) o formato no qual os dados são armazenados e (iii) as operações para armazenar e recuperar dados nos blocos.

Como um simples fluxo de *bytes* é a forma mais comum em que dados são dispostos no disco, memória ou trafegados através de uma rede, este é o formato mais adequado para representá-los. Desta forma, um bloco é definido como um vetor de *bytes*. O comprimento desse vetor, chamado *tamanho do bloco*, é variável e depende de cada instância de um *Workspace*. O *framework* define os primeiros *bytes* de cada bloco como uma seção de metadados, denominada *header*, como ilustrado pela Figura 4.5.

Figura 4.5: Estrutura do Bloco
Fonte: O autor

Na Figura 4.5, o campo *node type* é um valor inteiro utilizado para determinar de qual estrutura de dados o bloco faz parte. Durante o processo de criação do bloco, este campo é nulo. Quando uma estrutura de dados requisita um novo bloco, ela atribui um valor ao *node type* e o bloco torna-se parte desta estrutura. A partir de então, este campo torna-se imutável, até que o bloco seja deletado. Quando um bloco é deletado, seu *node type* é novamente zerado pelo *framework* e ele pode ser inserido em uma lista de blocos livres, estando disponível para ser novamente utilizado. O campo *node type* garante que uma estrutura de dados não seja capaz de manipular blocos pertencentes à outras estruturas de dados.

Como blocos são utilizados em estruturas de índice encadeadas ou baseadas em árvores, os campos *previous page ID* e *next page ID* apontam para blocos vizinhos, no mesmo nível, formando uma lista circular duplamente encadeada. Estas referências constituem uma maneira útil de navegar entre os blocos de um mesmo nível, em ambas as direções. O fato de ser circular garante que sempre haverá um bloco de destino durante a navegação. Além disso, a concatenação de blocos, causada pela remoção de objetos, é facilitada, devido a lista encadeada. Finalizando a seção *header* do bloco, os *bytes* livres remanescentes são usados pelas estruturas de dados alocarem suas próprias informações e dados.

Embora a identidade e meios de navegação entre os blocos estejam muito bem definidos, ainda é necessário prover as operações básicas para recuperar e armazenar dados (*bytes*) nos blocos. De acordo com a modelagem dos blocos, a forma mais conveniente de realizar este procedimento é encapsular o bloco em uma estrutura que forneça estas operações. A estrutura referida, denotada *Node*, é modelada na Figura 4.6.

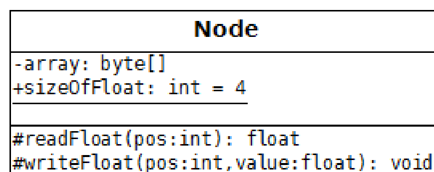


Figura 4.6: Modelagem do Node
Fonte: O autor

Como mencionado, a classe *Node* encapsula um bloco e fornece as operações para manipular e gerenciar *bytes*. O atributo *array* representa o bloco no qual os dados serão armazenados e recuperados. O *Node* implementa as operações de leitura e escrita para *todos* os tipos primitivos disponíveis para representar os dados.

Na Figura 4.6, as operações para ler e escrever números de ponto flutuante (tipo primitivo `float`) são exemplificadas. O método *readFloat* recebe como parâmetro um valor inteiro (*pos*) e, iniciando na posição *pos*, recupera e devolve um valor `float` para seu chamador. A partir da posição *pos*, o número de *bytes* utilizados para representar um valor `float` é dependente da arquitetura e definido pela constante *sizeOfFloat*. De forma similar, o método *writeFloat* recebe como parâmetros a posição *pos* e um valor `float` (*value*). Assim, este método converte o parâmetro *value* para sua representação no formato de *bytes* e o escreve no vetor, iniciando pela posição *pos* e ocupando *sizeOfFloat bytes* do vetor. De maneira semelhante, essas operações de leitura e escrita são definidas para todos os tipos primitivos.

Não é responsabilidade de um *Node* ou de seus métodos verificar a consistência dos dados. O *Node* não sabe como os dados são organizados e onde estão armazenados. Quando requisitado a ler um valor, um *Node* lerá o número de *bytes* especificado pelo tipo deste valor, os converterá de *bytes* para seu tipo original e irá devolvê-lo como um tipo primitivo. Se o valor devolvido faz ou não algum sentido para uma aplicação, é responsabilidade das estruturas de dados, que devem garantir que os *bytes* sejam lidos e escritos em suas posições corretas e nos tipos primitivos corretos.

Estruturas de dados distintas podem organizar um *Node* de diferentes formas. A fim de permitir que as estruturas de dados customizem os *Nodes* e sejam capazes de identificar a qual estrutura em particular um *Node* pertence, é necessário abstrair a classe *Node* de forma que ela possa ser especializada. Desta maneira, propõe-se uma hierarquia de classes, como mostrado na Figura 4.7.

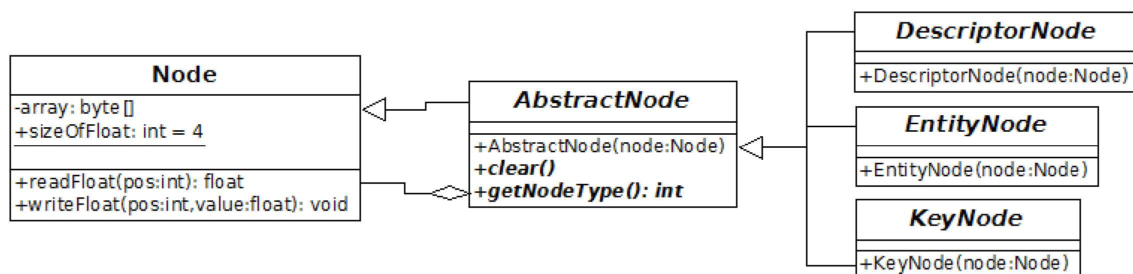


Figura 4.7: Hierarquia do módulo de Blocos
Fonte: O autor

A estrutura hierárquica da Figura 4.7 segue o padrão *Decorator*. Este padrão fornece uma alternativa flexível de extensão de funcionalidades pelas subclasses. A classe *Node* atua como o 'componente', definindo uma interface para objetos que podem ter responsabilidades adicionadas a si dinamicamente. Como um *Node* pode ser instanciado, ele também é um 'componente concreto', definindo um objeto no qual responsabilidades adicionais podem ser acopladas. A classe *AbstractNode* é o 'decorador'. Ela é uma referência para o 'componente' e fornece uma interface que segue a definição de um *Node*. As subclasses concretas de *AbstractNode* são os 'decoradores concretos' e adicionam responsabilidades ao 'componente'.

Diferente da especificação original do padrão *Decorator*, nesta modelagem, apenas um 'decorador concreto' pode decorar o 'componente'. Uma vez que 'decoradores concretos' são nós de estruturas de dados distintas, um mesmo nó não pode ser convertido para mais de um 'decorador concreto'. Quando um objeto *Node* é decorado, o construtor da classe *AbstractNode* verifica se o campo *node type* deste objeto e o campo *node type* do 'decorador concreto' são iguais. Se forem, o objeto *Node* passa a se comportar como o 'decorador concreto'; caso contrário, o processo não se completa. Para garantir essa funcionalidade, o construtor de *AbstractNode* é um *Template Method* e o método *getNodeType* é a 'operação primitiva', que obtém o valor do *node type*.

Na classe *AbstractNode*, o método *clear* é responsável por apagar a seção de cabeçalho, ou, ao menos, o campo *node type* quando um bloco (*Node*) é deletado. Esta operação é invocada a partir do método *deletePage* na classe *Workspace* (Seção 4.6).

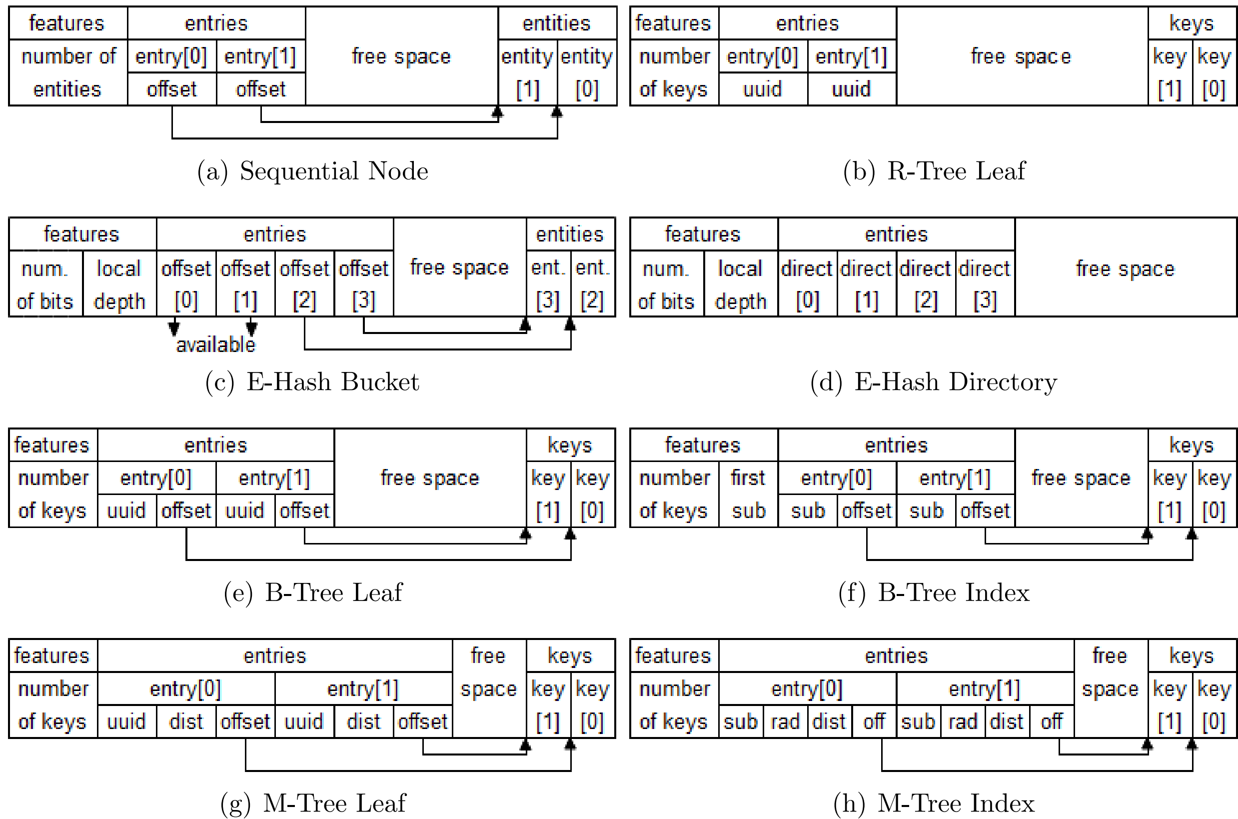
Como pode ser visto na Figura 4.7, a classe *AbstractNode* é especializada em *DescriptorNode*, *EntityNode* e *KeyNode*.

Um *DescriptorNode* é um nó especial usado pelas estruturas de dados armazenarem informações tais como a posição do nó raiz, a lista de blocos livres, valores estatísticos e a altura da árvore. Esta classe é derivada para cada estrutura de índice, como *SequentialDescriptor*, *EHashDescriptor*, *BTreeDescriptor*, *MTreeDescriptor* e *RTreeDescriptor*, todas omitidas na Figura 4.7. As classes *EntityNode* e *KeyNode* armazenam, respectivamente, as entidades persistentes e chaves indexadas. Ambas podem ser parametrizadas, garantindo que somente o tipo de objeto definido no parâmetro *template* será armazenado no bloco.

Os dados são organizados em cinco seções no interior dos blocos: (i) *header*, já descrita; (ii) *features*, armazenando dados gerais sobre os blocos; (iii) *entries*, com informações sobre os objetos; (iv) *entities/keys*, que são os objetos e (v) *free space*, que é uma área de espaço disponível no bloco. A personalização destas seções por cada estrutura de dados é ilustrada na Figura 4.8 e descrita a seguir. A seção *header* (Figura 4.5) encontra-se omitida em todas as ilustrações da Figura 4.8.

4.5.1 Nós da estrutura *Sequential*

SequantialNodes armazenam entidades persistentes nos blocos da estrutura *Sequential*. Um *SequentialNode* deriva da classe *EntityNode* e suas informações são apresentadas de acordo com a Figura 4.8(a). Nesta figura, a seção *features* retém informações particulares sobre o bloco, como o número de entidades ou objetos presentes no nó. Cada objeto armazenado em um Nó Sequencial tem um campo *offset* na seção *entries*. O *offset* referencia o primeiro *byte* do objeto, armazenado no fim do bloco. Entre as seções *entries* e *entities* há uma área de espaço livre, usada na inserção de novos objetos.



(a) Sequential Node
(b) R-Tree Leaf
(c) E-Hash Bucket
(d) E-Hash Directory
(e) B-Tree Leaf
(f) B-Tree Index
(g) M-Tree Leaf
(h) M-Tree Index

Figura 4.8: Organização e customização dos blocos
Fonte: O autor

Uma vez que os objetos possuem tamanhos variáveis, todas as informações de tamanho fixo são armazenadas no início do nó, enquanto que as informações de tamanho variável ficam no final. Assim, através do *offset*, um objeto pode ser rapidamente localizado, sendo esta a forma mais conveniente de fornecer acesso direto a objetos de tamanho variável dentro do nó. Esta abordagem, que é parecida com o gerenciamento de *split* da Árvore X (BERCHTOLD et al., 1996), torna-se prática e elegante, pois um nó não está restrito a um número pré-definido de objetos, mas apenas ao seu tamanho e espaço livre. Durante o processo de inserção, deve haver espaço livre suficiente para acomodar um novo objeto e uma entrada para esse objeto. Caso não haja, as estruturas de dados devem promover a divisão do nó, de acordo com suas próprias políticas de *split*.

4.5.2 Nós da estrutura *EHash*

Os nós de um *Hash* Extensível são especializados em *buckets* (folhas) e *directories* (índices) e derivam de *EntityNode*.

Os nós do tipo *EHashBucket* (Figura 4.8(c)) estão no último nível da estrutura de índices e armazenam as entidades persistentes. Na seção *features*, o campo *number of bits* determina o

número de entradas no bloco, que é sempre uma potência de 2 (para 3 *bits*, $2^3 = 8$ entradas). O campo *local depth* indica a posição de início de contagem de *bits*, com relação à quantidade total de bits da estrutura. A seção *entries* armazena o *offset* para os objetos, porém, como nem todas as posições podem ter sido mapeadas para objetos, alguns *offsets* podem estar disponíveis com valor nulo. Do mesmo modo que na estrutura Sequencial, entre as seções *entries* e *entities* há uma área de espaço livre.

Os nós do tipo *EHashDirectory*, mostrados na Figura 4.8(d) armazenam apenas as entradas de navegação para os sub-níveis da estrutura. Essas entradas são as referências de blocos em disco, sendo representadas por valores inteiros longos (tipo `long`). A seção *features* possui os mesmos campos descritos no *EHashBucket*.

4.5.3 Nós da *B-Tree*

Os nós da Árvore B derivam diretamente da classe *KeyNode* e armazenam somente as chaves indexadas. Estes nós são especializados em nós folha (*leaf*) e nós internos, também chamados de nós índices (*index*). Um nó folha (*BTreeLeaf*) armazena as chaves inseridas na Árvore B, enquanto que os nós índices (*BTreeIndex*) armazenam os objetos de roteamento para alcançar essas chaves.

Os nós do tipo *BTreeLeaf* são ilustrados pela Figura 4.8(e). Nestes blocos, a seção *features* contém o número de objetos inseridos no nó. Cada entrada da seção *entries* consiste de um *offset* e um *uuid*, usado para recuperar a instância de um objeto associado à sua chave. As seções *entries* e *keys* são separadas por uma região de espaço disponível.

Os nós do tipo *BTreeIndex* são mostrados na Figura 4.8(f). Ao invés de armazenar um *uuid*, cada entrada da seção *entries* guarda um valor chamado *sub*. Este valor referencia a raiz de uma sub-árvore, que pode ser outro nó índice ou um nó folha. Esta sub-árvore contém as chaves de valor maior ou igual à chave de sua entrada. Como os nós índice de uma Árvore B têm uma unidade a mais de referências que de chaves, o campo *first sub*, na seção *features*, referencia a sub-árvore cujas chaves são menores que a primeira chave (key[0]).

4.5.4 Nós da *R-Tree*

Os nós da Árvore R também são derivados em nós folha (*leaf*) e nós índices (*index*). Ambos são especializações da classe *KeyNode*.

A Figura 4.8(b) mostra um nó folha (*RTreeLeaf*). Diferentemente das outras estruturas de dados, as chaves de uma Árvore R não apresentam tamanho variável. Uma vez que as chaves são representadas pelo MBR, cujo tamanho em *bytes* é fixo, elas sempre terão o mesmo tamanho. Assim, como não é necessário armazenar um *offset* para localizar a chave (sua

posição pode ser diretamente calculada), cada entrada armazena somente um *uuid*. Embora a união do *uuid* e uma chave da Árvore R ainda tenham tamanho fixo, em *bytes*, os *uuids* e as chaves estão separados, pois os *uuids* não são parte da chave.

Os nós *RTreeIndex* são idênticos aos nós *RTreeLeaf*, exceto pelo campo *uuid*. O *uuid* é substituído pelo campo *sub*, apontando para uma sub-árvore, tal qual nos *BTreeIndex*.

4.5.5 Nós da *M-Tree*

O armazenamento de informações nos nós da Árvore M segue a mesma organização apresentada nos nós das outras árvores. A principal diferença entre os nós de uma Árvore M para as demais estruturas é a seção *entries*. As entradas, em um nó *MTreeLeaf*, como mostrado na Figura 4.8(g), têm seus campos ajustados às particularidades da Árvore M. No caso, uma folha contém um *uuid*, a distância para o nó representativo (*dist*) e um *offset*. De forma similar à Árvore B, as chaves da Árvore M podem ter tamanhos variáveis.

Nos nós do tipo *MTreeIndex* (Figura 4.8(h)), as entradas contêm o 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*.

4.6 O módulo de Dispositivos

O módulo de Dispositivos isola as estruturas 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 dado usado na sua persistência.

Uma vez que diferentes tipos de dispositivos apresentam características particulares, a maneira mais natural de representá-los é através de uma hierarquia de classes abstratas. Então, cada tipo de dispositivo deriva dessa hierarquia abstrata e implementa as operações necessárias. Dessa forma, torna-se fácil a adaptação e representação de outros meios de armazenamento. A hierarquia proposta é ilustrada pela Figura 4.9.

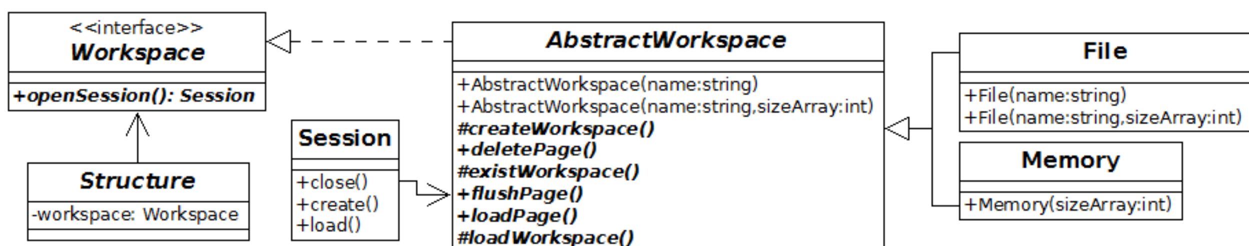


Figura 4.9: Hierarquia do módulo de Dispositivos
Fonte: O autor

Na Figura 4.9, a classe *AbstractWorkspace* é a interface comum a partir da qual todos os tipos de dispositivos de armazenamento são especializados. Ela representa uma área virtual que as estruturas de dados podem usar para acessar e compartilhar dados. Seus construtores comportam-se como *Template Methods*. Seguindo este padrão, os métodos *createWorkspace*, *existWorkspace* e *loadWorkspace* são as 'operações primitivas'. Estes métodos são responsáveis, respectivamente, por criar, verificar a existência e carregar um *workspace* de um dispositivo de armazenamento. Além disso, eles inicializam todos os atributos das subclasses de *AbstractWorkspace* antes que elas possam ser utilizadas, o que garante que os dispositivos de armazenamento estarão prontos para funcionamento. Esses três métodos são sobrescritos pelas subclasses de *AbstractWorkspace*, de forma a implementar os passos específicos para cada dispositivo de armazenamento.

A classe *AbstractWorkspace* é especializada nas classes *File* e *Memory*. A primeira manipula dados em um meio persistente e esses dados são mantidos salvos mesmo após o encerramento da aplicação, enquanto que a segunda manipula dados voláteis, que são perdidos quando a aplicação é encerrada.

Ainda na classe *AbstractWorkspace*, os métodos *loadPage* e *flushPage* representam as operações básicas para ler e escrever dados nos dispositivos de armazenamento. A respeito da operação *deletePage*, uma vez que um bloco é salvo em um *File* seus dados permanecerão no dispositivo de armazenamento para todo o sempre, ao menos, em teoria. A única forma de tornar novamente disponível seu espaço no dispositivo é invocando esse método.

A classe *Session* atua como uma área temporária de salvamento, mantendo as páginas recentemente utilizadas em uma estrutura de *hash*, em memória, para acelerar o acesso às mesmas. Nesta classe, o método *close* finaliza uma operação e envia os blocos manipulados na sessão para os dispositivos de armazenamento, invocando o método *flushPage* do *AbstractWorkspace*. O método *close* sempre é chamado após uma operação de inserção ou consulta, liberando os recursos computacionais alocados à essas operações.

O método *create* é utilizado pelas estruturas de dados alocarem espaço para uma nova página. Toda página retornada pelo método *create* é marcada com um identificador único e sequencial, chamado *page ID*. O *page ID* indica a posição da página (ou bloco) em seu *AbstractWorkspace*. Este método comporta-se como um *Factory Method*. Neste padrão, a classe *Session* é o 'criador' e devolve a instância de um *Node* (o 'produto'). O 'produto' define a interface dos objetos criados pelo *Factory Method*.

O método *load* é usado para recuperar blocos, armazenados no *AbstractWorkspace*, através de seu *page ID*. Em um primeiro momento, este método procura pelo bloco solicitado na estrutura de *hash* da *Session*. Se o bloco estiver presente na sessão atual, ele é recuperado em tempo constante; caso contrário, ele é carregado do *AbstractWorkspace* pelo método *loadPage*. Manter na sessão os blocos que ainda não foram liberados traz um ganho de

desempenho, uma vez que carregá-los a partir de um *File* é bem mais lento que recuperá-los a partir da memória RAM.

A *Session* é um objeto intermediário que garante que as estruturas de dados não acessarão os métodos de um *AbstractWorkspace* diretamente. Para este propósito, a classe *Workspace*, no topo da hierarquia, declara o método *openSession*, que devolve a instância de uma *Session*. Através dessa instância de *Session*, uma estrutura de dados torna-se apta a realizar operações com dispositivos de armazenamento, independente do *Workspace* (*File* ou *Memory*), sem conhecer a identidade deste *Workspace* e sem saber como essas operações são implementadas nestes *Workspaces*. O método *openSession* é outro *Factory Method*. Seguindo este padrão, aqui, a classe *Workspace* é o 'criador' e um objeto *Session* é o 'produto'.

Por fim, o módulo de Dispositivos está diretamente ligado ao módulo de Armazenamento através do padrão *Bridge*. Este padrão desacopla estes dois módulos, de forma que ambos possam variar independentemente. De acordo com o padrão *Bridge*, a classe *AbstractWorkspace* é o 'implementador', a classe que define as operações primitivas e fornece uma interface para a implementação de subclasses. As classes derivadas de *AbstractWorkspace*, *File* e *Memory*, são os 'implementadores concretos', definindo as implementações concretas. Finalmente, do lado do módulo de Armazenamento, a classe *Structure* é a 'abstração', mantendo a referência de uma instância de *AbstractWorkspace*, o 'implementador'. 'Abstrações' fornecem operações de alto nível, baseadas nas primitivas do 'implementador'. O padrão *Bridge* traz outra consequência: uma vez que as operações sobre dispositivos são encapsuladas e ocultadas pela classe *Workspace*, as estruturas de dados, derivadas de *Structure*, não sabem como essas operações são realizadas, mas podem acessá-las através da ponte estabelecida entre os dois módulos.

4.7 Trabalhos similares

Apesar da existência de estudos e trabalhos como Mattsson e Bosch (2000), Kienzle e Romanovsky (2002), Leist e Zellner (2006) e Oliveira et al. (2012), que contribuem para o desenvolvimento de *frameworks*, poucos deles focam em indexação, persistência, ou ambos.

Um dos primeiros trabalhos focados em persistência de registros foi o sistema Volcano (GRAEFE, 1994). Trata-se de uma aplicação fortemente acoplada em C++ que implementa um mecanismo de persistência e processamento de consultas. Esse sistema também provê operadores básicos como seleção e junção e a possibilidade do desenvolvedor criar seus próprios operadores, apesar disso não ser trivial, devido ao forte encapsulamento do sistema. Inicialmente, esse sistema não era focado na indexação de registros, mas, posteriormente, passou a utilizar uma Árvore B+ para indexar objetos. O mecanismo de consultas era baseado apenas em consultas pelos atributos dos registros armazenados.

A biblioteca GiST (HELLERSTEIN et al., 1995), também escrita em C++, implementa uma estrutura de indexação denominada *Generalized Search Tree*. Esta estrutura, baseada na lógica de *split* e *promote* das Árvores B e R, permite sua customização para indexar diferentes tipos de dados, sejam eles dados uni ou multidimensionais. A biblioteca também oferece um mecanismo de consulta simples, onde o processamento é baseado nos atributos dos objetos armazenados, embora não faça a persistência dos mesmos. Apesar de ter sido descontinuado, o GiST foi um projeto inovador em sua época, gerando publicações também em outras áreas, como compiladores e sistemas distribuídos.

O trabalho de Araújo et al. (1998), embora não seja um *framework* ou *library*, apresenta um editor de esquemas para mapeamento de objetos do banco de dados Sirius para bancos de dados relacionais, onde é realizada a persistência. Este trabalho não implementa estruturas de indexação e serialização de objetos; ambos são realizados pelo banco relacional.

O trabalho de Camargo et al. (2003) introduz um *framework* que explora o uso da Programação Baseada em Aspectos (AOP - *Aspect-Oriented Programming*) para auxiliar a persistência de classes Java em bancos de dados relacionais. Este *framework* é intrusivo, pois obriga as classes de aplicação do usuário a herdarem seu mecanismo de persistência. Uma vez que esta aplicação faz uso de banco relacional e, consequentemente, das propriedades ACID, ela não apresenta flexibilidade de armazenamento de dados. Entretanto, a modelagem através da AOP tornou o conjunto de classes deste *framework* fracamente acoplado e facilmente extensível.

Outra abordagem para a persistência de objetos é apresentada nos trabalhos de Weske e Kuropka (2001) e Alia et al. (2004). Estes trabalhos discutem e propõem estratégias e soluções de *frameworks* para persistência em ambientes *middleware*.

Outro projeto, o Arboretum (GBDI, 2005), é um *framework* escrito em C++ que fornece diversas implementações de estruturas de dados métricas, como as Árvores M (Seção 2.2.1), Slim (Seção 2.2.2) e outras. Este *framework* implementa facilidades para utilizar, desenvolver e testar estruturas métricas e operações de busca por similaridade. Embora algumas de suas abstrações de indexação tenham inspirado a modelagem de classes do *framework* Object-Injection, o Arboretum não provê um mecanismo de identificação única de objetos e sua indexação é restrita ao espaço métrico. Outra ausência deste *framework* é a capacidade de separar os objetos do usuários em entidades persistentes e chaves indexadas.

Ainda seguindo a linha da Programação Baseada em Aspectos, Kienzle e Gélineau (2006) reimplementaram as propriedades ACID usando AOP e aplicaram esta implementação na persistência de objetos transacionais.

Algumas soluções alternativas para persistência, largamente empregadas, são os *frameworks* de Mapeamento Objeto-Relacional (ORM - *Object-Relational Mapping*). Estes *frameworks* fazem a conversão dos dados de um domínio orientado a objetos para o modelo relacional,

para que possam ser persistidos em bancos de dados relacionais. Embora o Hibernate (BAUER; KING, 2004; JBOSS, 2012) seja o mais famoso e difundido, existem vários outros, como a JPA (*Java Persistence API*) (JSR-220, 2012) e o JDO (*Java Data Objects*) (JSR-12, 2012).

O trabalho de Batko et al. (2007) apresenta um *framework* chamado Messif. Trata-se de uma aplicação Java, de código fechado. Este trabalho implementa estruturas de indexação de dados voltadas para o espaço métrico, como as Árvores M e Slim. Seus autores descrevem o Messif como um *framework* modularizado e facilmente extensível, facilitando também o desenvolvimento de algoritmos de consulta por similaridade. Contudo, este trabalho não realiza persistência de objetos.

O *Extensible and Flexible Framework* (XXL) (BERCKEN et al., 2000; BERCKEN et al., 2001; CAMMERT et al., 2003; XXL, 2012) é, atualmente, o trabalho com ideias mais similares ao *framework* Object-Injection. Trata-se de uma aplicação de alto nível, escrita em Java e de código aberto, cujo principal objetivo é dar suporte à criação de algoritmos para o processamento de consultas. Este *framework* implementa uma coleção de estruturas de indexação, como as Árvores B+, M e R, e operadores de busca, para facilitar o desenvolvimento de novos algoritmos de consulta. Com um mecanismo de persistência fortemente baseado nas classes nativas da linguagem Java, os usuários devem estender seu modelo de classes e reescrever os métodos necessários para persistir objetos, o que faz do XXL um *framework* invasivo. Além disso, a modelagem de classes do *framework* XXL não é suficientemente clara para que o usuário saiba qual(is) classe(s) deve estender. Visto que o XXL manipula as classes de aplicação do usuário como *Object*, que é a raiz da hierarquia das classes Java, este *framework* evita redundância de informações. Além disso, ele não faz uso de bancos relacionais para persistir objetos.

Para realizar a persistência de objetos, muitas das citadas soluções usam mecanismos ou sistemas externos à aplicação principal, o que acaba por sobrecarregá-la. O *framework* Object-Injection difere dessas aplicações da literatura por ser facilmente acoplado às aplicações usuárias, fornecendo indexação e persistência de forma nativa e transparente. Além disso, o mecanismo de persistência implementado pelo Object-Injection não faz uso de bancos de dados relacionais e sua arquitetura de classes pode ser facilmente estendida para dar suporte ao desenvolvimento de novas estruturas de dados ou algoritmos de consulta.

Capítulo 5

Experimentos

Este Capítulo descreve os experimentos realizados para testar o *framework* Object-Injection. Visando avaliar o desempenho deste *framework*, foram realizados experimentos com o *framework* XXL, seguidos de uma comparação entre as duas soluções.

A Seção 5.1 descreve os conjuntos de dados utilizados nos testes, os parâmetros utilizados como referência para aferir os experimentos e as configurações adotadas em ambas as soluções testadas. A Seção 5.2 apresenta os gráficos confeccionados a partir dos resultados obtidos, discute as implicações e comportamentos concernentes a esses resultados e finaliza enfatizando os ganhos e perdas de ambos os *frameworks*. Esta Seção também faz alusão à distância utilizada nos métodos de acesso métricos.

5.1 Metodologia

Para realizar experimentos com o *framework*, foram utilizados três conjuntos de dados. O primeiro conjunto é formado por palavras extraídas dos dicionários do OpenOffice (OPENOFFICE, 2012). Esta base combina palavras de 14 diferentes idiomas: alemão da Alemanha, dinamarquês, esloveno, espanhol da Espanha, francês, holandês, húngaro, inglês dos EUA, italiano, norueguês (variante bokmål), norueguês (variante nynorsk), polonês, português do Brasil e sueco, totalizando 1.938.058 palavras. As especificações "alemão da Alemanha", "espanhol da Espanha", "inglês dos EUA" e similares referem-se ao arquivo de dados específico do país considerado; lembrando que vários idiomas possuem variantes em outros países, como alemão (língua oficial também na Áustria, Suíça e outros), espanhol (Espanha, Américas Andina e Central), inglês (Austrália, Canadá, EUA, Reino Unido etc), português (Brasil, Portugal, Angola, Timor Leste etc). Este conjunto de palavras foi pré-processado, de forma a eliminar duplicatas. As palavras deste conjunto de dados têm seu comprimento variando entre 2 a 41 caracteres.

O segundo conjunto de dados, obtido do Projeto UniProt (UNIPROT, 2012), consiste de 78.221 proteínas. Este conjunto de dados foi pré-processado, visando eliminar duplicatas e considerar apenas as cadeias de aminoácidos cujo comprimento é menor ou igual a 32 aminoácidos.

O terceiro conjunto de dados foi obtido do Projeto GEONet Names Server (GEONET, 2012), da *USA National Geospatial-Intelligence Agency*. Este conjunto contém informações sobre pontos geográficos (latitude e longitude) de localidades de todo o planeta. Esta base foi pré-processada para eliminar duplicatas, restando um total de 3.818.581 pontos distintos. Esses pontos encontram-se plotados na Figura 5.1.

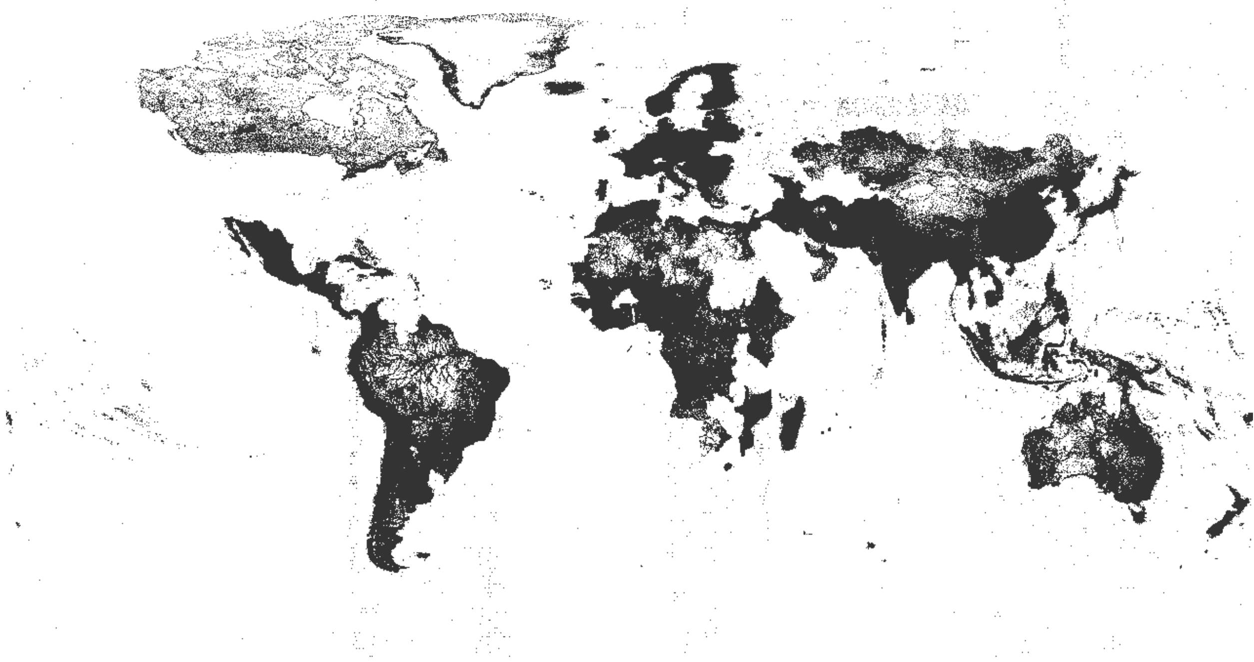


Figura 5.1: Distribuição dos pontos da base da dados GEONet
Fonte: O autor

Usando estes conjuntos de dados, os experimentos realizados mediram o tempo, o número de acessos a blocos e cálculos realizados nas operações de inserção e consulta pontual sobre índices secundários.

Os experimentos foram realizados num computador com processador Intel® Core™ i7-930, *clock* de 2.8 GHz, respectivamente 32 KB, 256 KB e 8 MB de *cache* L1, L2 e L3, com 4 GB de memória RAM e sistema operacional GNU Linux, distribuição Kubuntu 12.04 e 500 GB de HD SATA 7200 RPM. O *framework* Object-Injection foi implementado em Java, usando a OpenJDK 7.

Para avaliar o desempenho do *framework* Object-Injection, os testes foram conduzidos usando 4 tamanhos para os blocos de disco (2 KB, 4 KB, 8 KB e 16 KB) e o mesmo teste foi realizado sobre o *framework* XXL, também disponível em Java, para fins de comparação.

Como o *framework* XXL não implementa índices primários, os testes foram realizados apenas com os índices secundários (Árvores B, M e R).

Uma vez que a persistência do XXL é fortemente baseada na serialização Java, uma das dificuldades com este *framework* foi ajustar o tamanho dos blocos. No *framework* XXL, o tamanho dos blocos é definido através de uma relação entre as quantidades mínima e máxima de objetos armazenados no bloco. Quando estes objetos têm tamanhos fixos, em *bytes*, como no terceiro conjunto de dados, esta relação garante a ocupação máxima do bloco. Contudo, quando as chaves têm tamanho variante, foi utilizado o tamanho médio das chaves, em *bytes*, como parâmetro de ajuste desta propriedade. No *framework* Object-Injection não há necessidade de realizar este ajuste, visto que as chaves são inseridas nos blocos enquanto houver espaço disponível, como descrito na Seção 4.5.

O *framework* XXL utiliza uma área de *buffer* para armazenamento temporário dos blocos manipulados. O tamanho desta região foi configurado para ter a mesma altura das árvores de indexação, visando reduzir o ganho proporcionado por este recurso. O *framework* Object-Injection utiliza uma técnica similar, a *Session* (Seção 4.6), mas este mecanismo é limpo e reinicializado para cada operação de inserção ou consulta realizada. Ambas as técnicas auxiliam na navegação *bottom-up* e *top-down* das árvores.

5.2 Resultados

O primeiro conjunto de dados (OpenOffice) foi inserido em uma Árvore B+. Neste experimento, para satisfazer a relação de ordem total, foi considerado a ordem alfabética entre as palavras. Os resultados obtidos são mostrados na Figura 5.2, onde, na legenda, a sigla "OI" refere-se ao *framework* Object-Injection e a sigla "XXL" ao *framework* XXL.

Como pode ser visto na Figura 5.2(a) (tempo médio de inserção) e na Figura 5.2(b) (tempo médio para consulta), o comportamento de ambos os *frameworks* é estável, com as linhas dos gráficos (tempo) acompanhando o crescimento do tamanho do bloco.

A Figura 5.2(c) apresenta o número médio de acessos a blocos, na inserção. Nesta medição, o *framework* Object-Injection gerencia seus blocos usando as classes descritas na Seção 4.6, enquanto que o XXL delega esta responsabilidade para as extensões do mecanismo de serialização Java, sobre o qual estes resultados foram aferidos.

Na Figura 5.2(d) (média de acesso a blocos na consulta), pode-se perceber que, para blocos de 2 KB e 4 KB, o número de acessos a disco é o mesmo (4 acessos). Isto ocorre porque ambas as árvores, construídas pelo Object-Injection, têm a mesma altura, porém as folhas dos blocos de 2 KB contém um número maior de objetos que as folhas dos blocos de 4 KB. Este mesmo comportamento repete-se com os blocos de 8 KB e 16 KB (3 acessos).

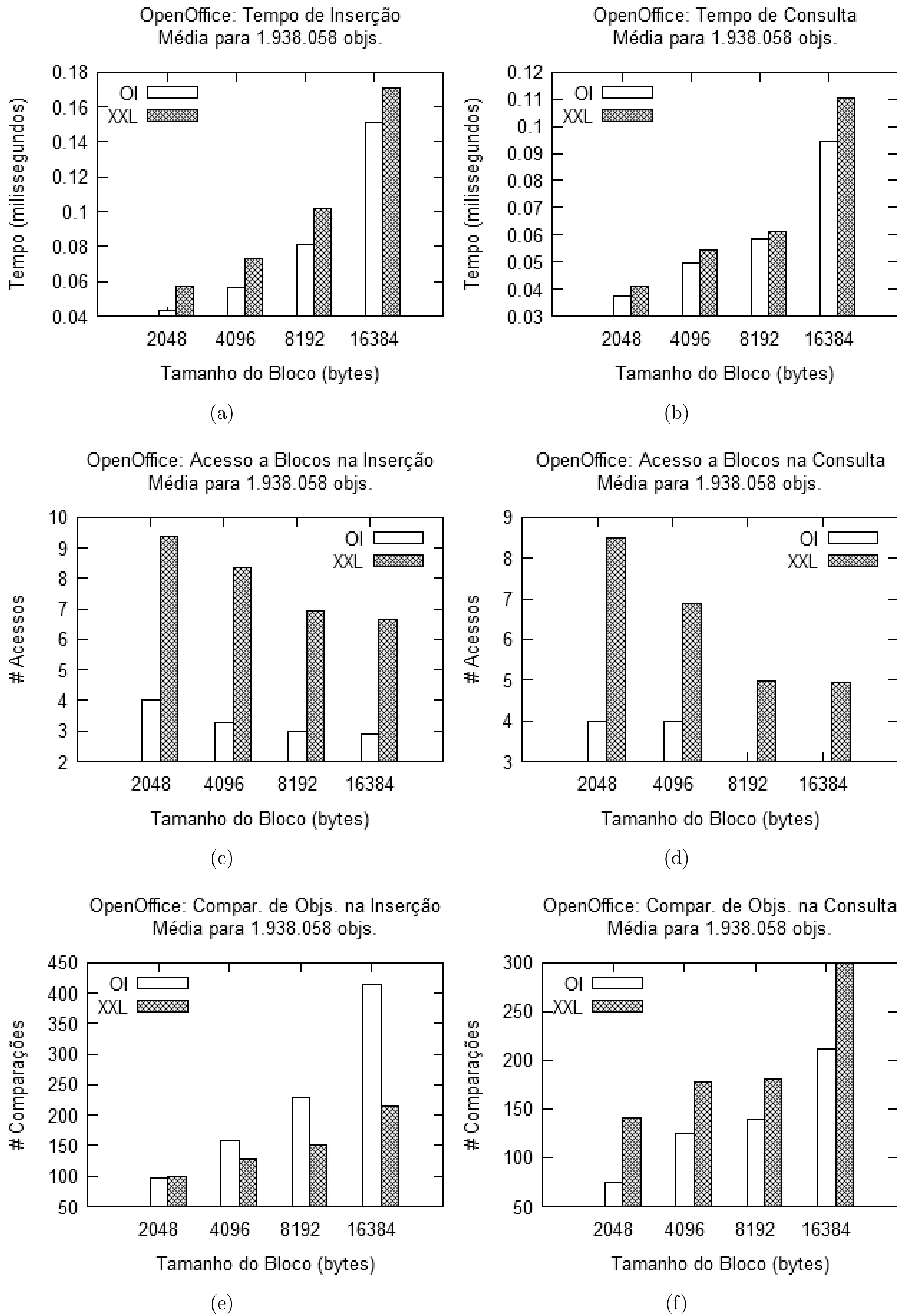


Figura 5.2: Experimento: conjunto de dados do OpenOffice

Sobre o número médio de comparações de objetos na inserção (Figura 5.2(e)), enquanto o Object-Injection utiliza o algoritmo *Insertion-Sort* (KNUTH, 1998) para adicionar e ordenar uma nova chave em um bloco, durante uma operação de *split*, o XXL usa a implementação do *Merge-Sort* (KNUTH, 1998) nativa do Java. Assim, espera-se que o *framework* Object-Injection realize um número maior de comparações que o XXL. O número médio de comparações de objetos, na operação de consulta, como mostrado pela Figura 5.2(f), não é afetado pelo uso dos algoritmos citados, visto que ambos os *frameworks* utilizam a Busca Binária para localizar a sub-árvore a ser percorrida. Contudo, de acordo com a maneira na qual o XXL gerencia seus blocos, sua árvore é maior (em altura) que a árvore do Object-Injection, razão pela qual o XXL realiza um maior número de comparações até alcançar uma folha.

O segundo conjunto de dados (UniProt) foi indexado em uma Árvore M. Como função de distância, ou métrica, foi utilizada uma variação (XU et al., 2003) da distância de Damerau-Levenshtein (DAMERAU, 1964), apresentada no Apêndice A e daqui em diante referida como métrica de Xu. A métrica de Xu combina o custo das operações de inserção, substituição, remoção e transposição com o custo da substituição de um aminoácido por outro, de acordo com a matriz mPAM (*Matrix Point Accepted Mutation*) (XU; MIRANKER, 2004). Xu et al. (2003) sugerem como penalidade inicial adotada nas inserções e remoções o maior valor da matriz mPAM, que é 7.

Em ambos os *frameworks*, o algoritmo de promoção aleatório (*random promote*) e o método de distribuição das chaves por hiperplano generalizado foram usados. Como a Árvore M original do *framework* XXL computa apenas distâncias entre pontos usando apenas a(s) métrica(s) L_p, ajustes foram realizados em alguns de seus métodos, de forma a utilizar a métrica de Xu.

Os resultados dos experimentos com a base UniProt são ilustrados pela Figura 5.3. A Figura 5.3(a) (tempo médio de inserção) e a Figura 5.3(e) (número médio de cálculos de distância na inserção) mostram linhas crescentes, como esperado, uma vez que, em blocos maiores, um maior número de cálculos de distância é realizado, devido ao maior número de chaves armazenadas. O tempo médio de inserção sofre maior influência do número de cálculos de distância (Figura 5.3(e)) do que da quantidade de acesso a blocos (Figure 5.3(c)).

As Figuras 5.3(b), 5.3(d) and 5.3(f) seguem uma mesma tendência, o que indica que as medidas de acesso a blocos e cálculos de distância têm igual influência sobre o tempo das operações de consultas pontual.

O terceiro conjunto de dados (GEONet) foi inserido em uma Árvore R e a Figura 5.4 mostra os resultados obtidos. Em ambos os *frameworks* foi usado o algoritmo de *split* quadrático na divisão dos nós. As coordenadas geográficas no XXL são representadas apenas por números de ponto flutuante de precisão dupla (tipo `double`). Este mesmo tipo de dado foi usado no Objetct-Injection, embora este *framework* suporte outros tipos.

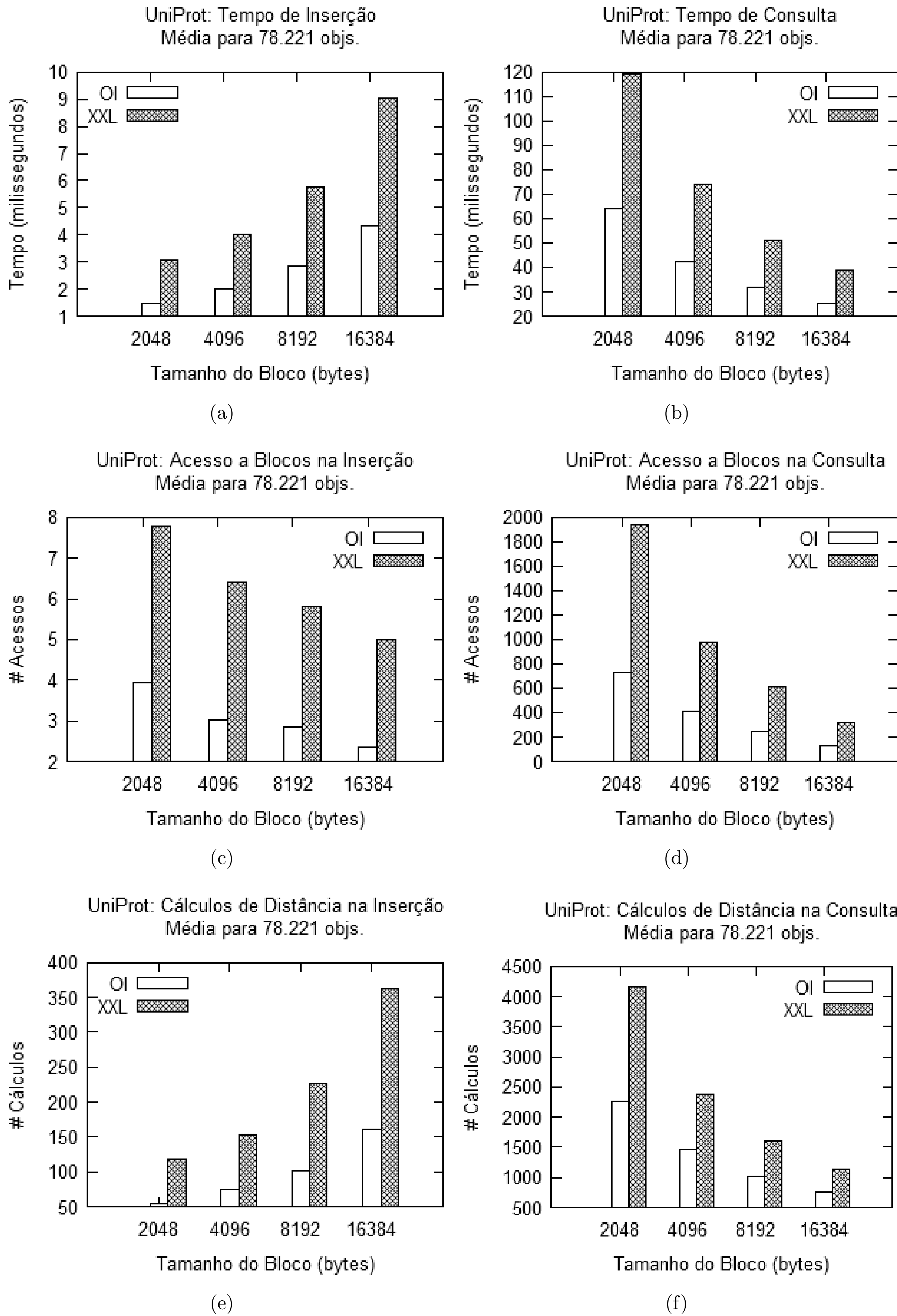


Figura 5.3: Experimento: conjunto de dados UniProt

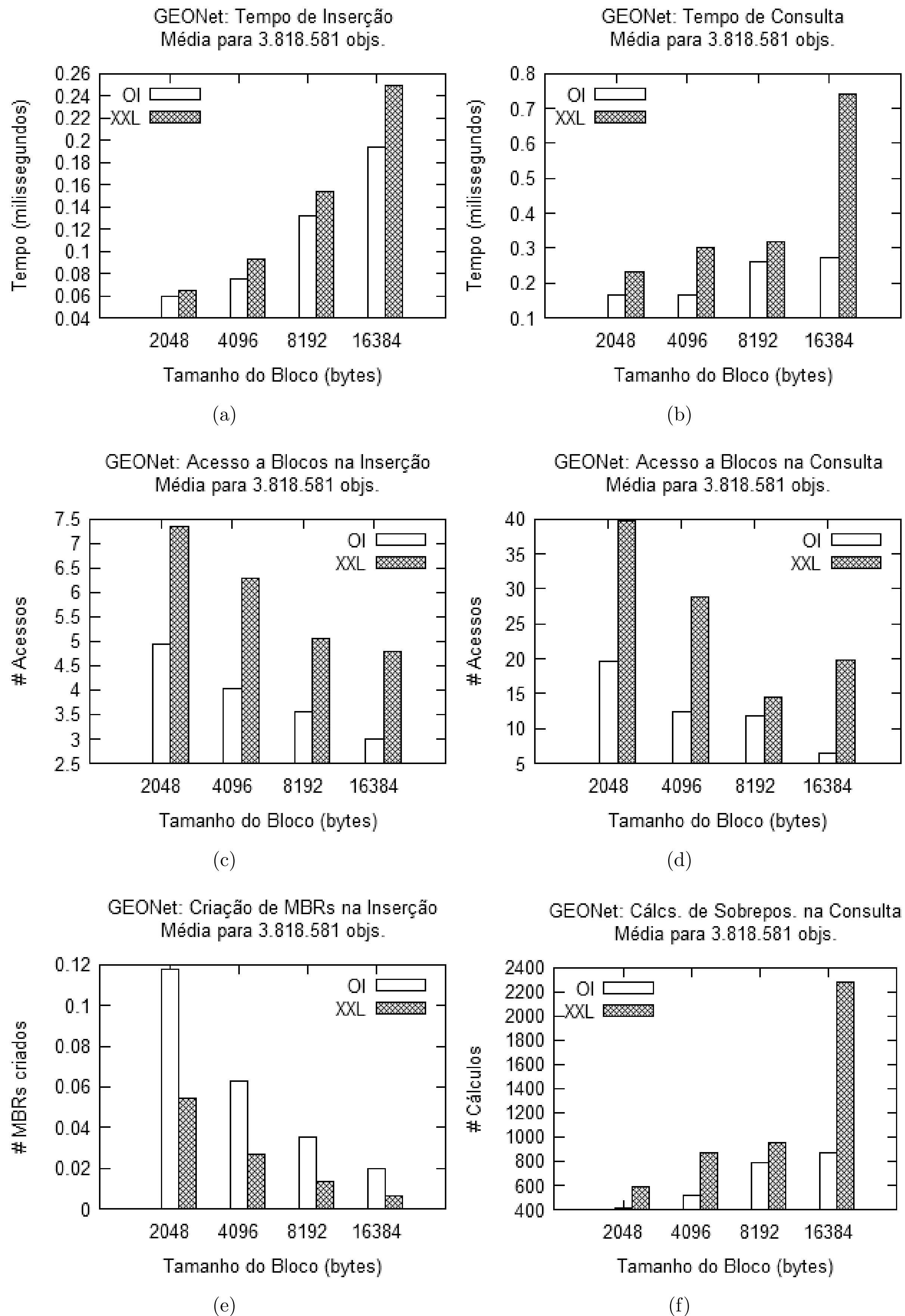


Figura 5.4: Experimento: conjunto de dados GEONet

A Figura 5.4(a) mostra o tempo médio de inserção e, como esperado, este tempo cresce conforme o tamanho de bloco aumenta, devido à uma maior quantidade de chaves que podem se qualificar como sub-árvores a serem percorridas. Esse mesmo comportamento pode ser notado na Figura 5.4(b) (tempo de consulta), mas de forma mais suave, visto que o algoritmo de *split* quadrático gera uma menor área de sobreposição entre os MBRs. Apesar do *framework* Object-Injection criar um maior número de MBRs durante a inserção (Figura 5.4(e)), a área de intersecção entre esses MBRs é bem menor que no XXL, como pode ser visto na Figura 5.4(f), que computa a média do cálculo das intersecções nas consultas.

Na Figura 5.4(c), o número médio de acessos a blocos, na inserção, mantém-se estável em ambos os *frameworks*, mas a Figura 5.4(d) sugere que a Árvore R do XXL degenera conforme o tamanho dos blocos aumenta.

A Tabela 5.1 apresenta a quantidade de espaço em disco, em *bytes*, usada para armazenar os conjuntos de testes nas estruturas de dados de ambos os *frameworks*, com blocos de 2 KB, 4 KB, 8 KB e 16 KB. No armazenamento da Árvore B, o Object-Injection consome, em média, cerca de 22,36% menos espaço em disco. Contudo, na Árvore M e Árvore R, o XXL consome, respectivamente, 8,16% e 55,70% menos espaço em disco, em média. A razão desta perda é que os índices secundários do Object-Injection armazenam, adicionalmente, para cada chave, 16 *bytes* de UUID. Considerando a quantidade de dados em cada conjunto de teste, isso representa cerca de 31.008.928 *bytes* extras na Árvore B, 1.251.536 na Árvore M e 61.097.296 na Árvore R.

Tabela 5.1: Tamanho dos arquivos (*bytes*)

Bloco (*bytes*)	B-Tree OI	B-Tree XXL	M-Tree OI	M-Tree XXL	R-Tree OI	R-Tree XXL
2.048	101.103.616	130.791.004	7.823.360	6.882.120	317.896.704	211.649.770
4.096	99.864.576	128.168.082	7.593.984	6.951.338	322.007.040	207.556.988
8.192	99.901.440	127.242.028	7.397.376	6.906.068	324.845.568	205.871.244
16.384	100.106.240	130.271.172	7.127.040	6.946.922	329.990.144	206.654.546

Em uma análise geral, a Árvore B do Object-Injection é 19,42% mais rápida que a do XXL na inserção e 8,90% nas consultas, com relação a média dos tamanhos de bloco. Os resultados da Árvore M mostram um ganho de tempo médio de 51,23% na inserção e 40,43% nas consultas, do Object-Injection sobre o XXL. Com relação à Árvore R, o *framework* Object-Injection supera, em tempo, o XXL em 15,81% na inserção e 38,38% nas consultas.

Mesmo com os ganhos obtidos na Árvore R, o pior resultado computado para o Object-Injection foi com relação ao número de MBRs criados na inserção, onde o XXL gera, em média, uma quantidade 54,03% menor.

Capítulo 6

Conclusão

Os tempos atuais estão sendo marcados pela enorme quantidade de informações que deve ser processada e armazenada, fazendo com que os figurantes da sociedade sintam-se confusos. Uma solução para este problema é proposta pela área de Banco de Dados. Esta área, proeminente da Ciência da Computação, está crescendo cada vez mais, pois fornece mecanismos que facilitam a representação da informação, utilizando-se de recursos aprimorados, tanto em variedade de sistemas e técnicas empregadas, quanto em importância.

Ao longo deste trabalho, foram estudadas e implementadas estruturas de dados que realizam o armazenamento e indexação de objetos. Estas estruturas permitem o armazenamento dos dados de forma organizada, de modo a agilizar as operações de consulta aos objetos armazenados. Estruturas de índice primário, como a lista sequencial e o *hash* extensível, mostram-se apropriadas para o armazenamento massivo de grandes coleções de objetos. Por sua vez, as estruturas de índice secundário, como as árvores B+, R e M, auxiliam o processamento de consulta e recuperação de objetos armazenados, seja através de uma relação de ordem, posicionamento espacial ou através de uma medida de similaridade.

Além disso, também foram investigadas soluções de modelagem e otimização de *software*, conhecidas como padrões de projeto. O estudo dos padrões de projeto possibilitou uma visão clara de como podem ser implementados, de maneira estruturada, soluções reutilizáveis de *software* orientados a objetos, observando as vantagens, desvantagens e consequências do uso de cada padrão.

O objetivo principal deste trabalho foi consolidado pelo desenvolvimento de um mecanismo de indexação e persistência de dados, através da definição e implementação do *framework* orientado a objetos, chamado *Object-Injection*.

O *framework* Object-Injection não é dependente de uma técnica ou linguagem de programação específica, podendo ser implementado em qualquer linguagem orientada a objetos, ainda que este trabalho tenha feito uso da linguagem Java.

A modelagem do *framework* Object-Injection utiliza de CRTP e padrões de projeto. Isto possibilitou abstrair questões semânticas e de encapsulamento, além de permitir a extensão de suas funcionalidades para novos dispositivos e estruturas de dados. Dessa forma, conclui-se que o emprego destas duas técnicas, bem como a escolha dos padrões de projeto, foi acertada.

Visando promover uma fraca dependência e acoplamento entre as classes do *framework*, sua arquitetura foi dividida em quatro módulos, cada um sendo responsável por gerenciar tarefas ou componentes específicos da aplicação.

O módulo das Metaclasses define um modelo de classes hierárquico que deve ser seguido a fim de garantir a persistência para as classes de aplicação do usuário. Com respeito à indexação, este módulo ainda classifica as diversas relações entre os domínio dos dados, seja este domínio baseado na relação de ordem total, em uma métrica, ou em propriedades multidimensionais.

No módulo de Armazenamento são implementadas as estruturas de índices responsáveis por gerenciar os dados do módulo das Metaclasses. Diferentemente de outras soluções, como *frameworks* ORM, o módulo dos Blocos implementa seu próprio esquema de serialização de objetos, fazendo com que o mecanismo de persistência não necessite de SGBDRs.

O módulo dos Dispositivos constitui uma fábrica de blocos, vinculando os módulos de Armazenamento e Blocos. O módulo dos Dispositivos pode também ser estendido, permitindo que outros dispositivos ou meios de armazenamento sejam facilmente definidos.

Como é evidenciado pelos experimentos, o *framework* Object-Injection é uma solução completa e flexível, permitindo o armazenamento de objetos, com consideráveis ganhos de desempenho em relação à alternativas similares. Além disso, o *framework* apresenta um comportamento estável ao manipular dados multidimensionais e um significante e expressivo ganho de desempenho sobre o espaço métrico.

Como propostas futuras, pode-se citar a criação de um mecanismo, baseado em *anotações* de classes, que implemente automaticamente as interfaces de indexação e persistência do módulo das Metaclasses. Este mecanismo contribuiria para um aumento de produtividade do programador, uma vez que classes puramente operacionais, sem relação com a lógica de negócio, poderiam ser geradas de forma automatizada.

Inspirando-se no trabalho de Budíková et al. (2012), outra proposta futura seria a criação de um módulo de consulta aos objetos armazenados pelo *framework* Object-Injection. Este novo módulo seria responsável por implementar os diferentes algoritmos de consultas, tanto métricas como espaciais, e por definir a estrutura e sintaxe de sua linguagem de consulta.

Finalmente, formas descentralizadas de armazenamento poderiam oferecer maior escalabilidade e conectividade aos dados. Além disso, o armazenamento necessita de um suporte às propriedades ACID. Exemplos de novos meios de armazenamento seriam cliente-servidor, computação em grade e computação em nuvem.

Referências Bibliográficas

ALAGIC, S. The ODMG object model: does it make sense? In: **Proceedings of the 12th ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications**. Atlanta, GA, USA: ACM, 1997. (OOPSLA '97), p. 253–270.

ALIA, M.; BARRIOZ, S. C.; DECHAMBOUX, P.; HAMON, C.; LEFEBVRE, A. A middleware framework for the persistence and querying of java objects. In: **Proceedings of the 18th European Conference on Object-Oriented Programming**. Oslo, Norway: Springer, 2004. (Lecture Notes in Computer Science, v. 3086), p. 291–315.

ANGLES, R.; GUTIERREZ, C. Survey of graph database models. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 40, n. 1, p. 1:1–1:39, fev. 2008.

ARAÚJO, M. R. B.; TRAINA JR., C.; BIAJIZ, M.; MACHADO, E. P. Editor de esquemas com suporte para hierarquias de classificação. **Anais do 13º Simpósio Brasileiro de Banco de Dados**, SBBD, p. 1–11, 1998.

ASTRAHAN, M. M.; BLASGEN, M. W.; CHAMBERLIN, D. D.; ESWARAN, K. P.; GRAY, J. N.; GRIFFITHS, P. P.; KING, W. F.; LORIE, R. A.; MCJONES, P. R.; MEHL, J. W.; PUTZOLU, G. R.; TRAIGER, I. L.; WADE, B. W.; WATSON, V. System R: relational approach to database management. **ACM Transactions on Database System**, ACM, New York, NY, USA, v. 1, n. 2, p. 97–137, jun. 1976.

BARRY, D.; STANIENDA, T. Solving the java object storage problem. **IEEE Computer**, IEEE Computer Society Press, Los Alamitos, CA, USA, v. 31, n. 11, p. 33–40, nov. 1998.

BATKO, M.; NOVAK, D.; ZEZULA, P. Messif: metric similarity search implementation framework. In: **Proceedings of the 1st International Conference on Digital Libraries: Research and Development**. Pisa, Italy: Springer-Verlag, 2007. (DELOS '07, v. 4877), p. 1–10.

BAUER, C.; KING, G. **Hibernate in action**. Greenwich, CT, USA: Manning Publications, 2004. 408 p. (In Action Series).

BAYER, R.; MCCREIGHT, E. Organization and maintenance of large ordered indexes. **Acta Informatica**, ACM, New York, NY, USA, v. 3, n. 1, p. 173–189, 1972.

BECK, K.; CUNNINGHAM, W. Using pattern languages for object-oriented programs. **Conference on Object-Oriented Programming Systems, Languages, and Applications**, ACM, Orlando, FL, USA, set. 1987.

BECKMANN, N.; KRIEGEL, H.-P.; SCHNEIDER, R.; SEEGER, B. The R*-tree: An efficient and robust access method for points and rectangles. **ACM SIGMOD Record**, ACM, New York, NY, USA, v. 19, n. 2, p. 322–331, maio 1990.

BERCHTOLD, S.; KEIM, D. A.; KRIEGEL, H.-P. The X-tree: An index structure for high-dimensional data. In: **Proceedings of the 22nd International Conference on Very Large Data Bases**. Mumbai, India: Morgan Kaufmann Publishers Inc., 1996. (VLDB '96), p. 28–39.

BERCKEN, J. v. d.; BLOHSFELD, B.; DITTRICH, J. P.; KRÄMER, J.; SCHÄFER, T.; SCHNEIDER, M.; SEEGER, B. Xxl - a library approach to supporting efficient implementations of advanced database queries. In: **Proceedings of 27th International Conference on Very Large Data Bases**. Roma, Italy: Morgan Kaufmann Publishers Inc., 2001. (VLBD '01), p. 39–48.

BERCKEN, J. v. d.; DITTRICH, J. P.; SEEGER, B. javax.xxl: a prototype for a library of query processing algorithms. **ACM SIGMOD Record**, New York, NY, USA, v. 29, n. 2, p. 588–590, 2000.

BOOST. **Boost C++ Libraries**. 2012. Disponível em: <http://www.boost.org>. Acesso em: 21 de novembro de 2012.

BRAKATSOULAS, S.; PFOSER, D.; THEODORIDIS, Y. Revisiting R-tree construction principles. In: **Proceedings of the 6th East European Conference on Advances in Databases and Information Systems**. Bratislava, Slovakia: Springer-Verlag, 2002. (Lecture Notes in Computer Science, v. 2435), p. 149–162.

BUDÍKOVÁ, P.; BATKO, M.; ZEZULA, P. Query language for complex similarity queries. In: **Proceedings of 16th East European Conference on Advances in Databases and Information Systems**. Pozna, Poland: Springer, 2012. (Lecture Notes in Computer Science, v. 7503), p. 85–98.

CAMARGO, V. V.; RAMOS, R. A.; PENTEADO, R. A. D.; MASIERO, P. C. Projeto baseado em aspectos do padrão camada de persistência. In: **Anais do 12º Simpósio Brasileiro de Engenharia de Software**. Manaus, AM, Brasil: SBES, 2003. p. 114–129.

CAMMERT, M.; HEINZ, C.; KRÄMER, J.; SCHNEIDER, M.; SEEGER, B. A status report on xxl - a software infrastructure for efficient query processing. **IEEE Data Engineering Bull.**, v. 26, n. 2, p. 12–18, 2003.

CAMPBELL, R. H.; ISLAM, N.; MADANY, P. Choices, frameworks and refinement. **Computing Systems**, v. 5, n. 3, p. 217–257, 1992.

CAREY, M. J.; DEWITT, D. J. Of objects and databases: A decade of turmoil. In: **Proceedings of the 22nd International Conference on Very Large Data Bases**. Mumbai, India: Morgan Kaufmann Publishers Inc., 1996. (VLDB '96), p. 3–14.

CATTELL, R. G. G.; BARRY, D. K. **The object data standard: ODMG 3.0**. 3rd. ed. San Francisco, CA, USA: Morgan Kaufmann, 2000. 288 p.

CHANG, F.; DEAN, J.; GHEMAWAT, S.; HSIEH, W. C.; WALLACH, D. A.; BURROWS, M.; CHANDRA, T.; FIKES, A.; GRUBER, R. E. Bigtable: A distributed storage system for structured data. In: **Proceedings of 7th Symposium on Operating System Design and Implementation**. Seattle, WA, USA: USENIX Association, 2006. (OSDI '06), p. 1–14.

CHÁVEZ, E.; NAVARRO, G.; BAEZA-YATES, R.; MARROQUÍN, J. L. Searching in metric spaces. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 33, n. 3, p. 273–321, set. 2001.

CIACCIA, P.; PATELLA, M. The M^2-tree: Processing complex multi-feature queries with just one index. In: **Proceedings of 1st DELOS Network of Excellence Workshop on Information Seeking, Searching and Querying in Digital Libraries**. Zurich, Switzerland: Online publication, 2000. p. 1–6. Disponível em: <`http://www.ercim.eu/publication/ws-proceedings/DelNoe01/`>. Acesso em: 8 de fevereiro de 2012.

CIACCIA, P.; PATELLA, M.; ZEZULA, P. M-tree: An efficient access method for similarity search in metric spaces. In: **Proceedings of 23rd International Conference on Very Large Data Bases**. Athens, Greece: Morgan Kaufmann Publishers Inc., 1997. (VLBD '97), p. 426–435.

CODD, E. F. A relational model of data for large shared data banks. **Communications of the ACM**, ACM, New York, NY, USA, v. 13, n. 6, p. 377–387, jun. 1970.

COMER, D. The ubiquitous B-tree. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 11, n. 2, p. 121–137, jun. 1979.

COPLIEN, J. O. A curiously recurring template pattern. **Cpp Report**, SIGS Publications Group, New York, NY, USA, v. 7, n. 2, p. 40–43, fev. 1995.

CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. **Introduction to Algorithms**. 3rd. ed. New York, NY, USA: The MIT Press, 2009. 1313 p.

DAMERAU, F. J. A technique for computer detection and correction of spelling errors. **Communications of the ACM**, ACM, New York, NY, USA, v. 7, n. 3, p. 171–176, mar. 1964.

DECANDIA, G.; HASTORUN, D.; JAMPANI, M.; KAKULAPATI, G.; LAKSHMAN, A.; PILCHIN, A.; SIVASUBRAMANIAN, S.; VOSSHALL, P.; VOGELS, W. Dynamo: amazon's highly available key-value store. In: **Proceedings of 21st ACM Symposium on Operating Systems Principles**. Stevenson, Washington, USA: ACM, 2007. (SOSP '07), p. 205–220.

DEUTSCH, L. P. Design reuse and frameworks in the smalltalk-80 system. In: BIGGERSTAFF, T. J.; PERLIS, A. T. (Ed.). **Software Reusability**. Reading, MA, USA: Addison-Wesley, 1989, (Applications and Experience, v. 2). p. 57–71.

DROZDEK, A. **Data Structures and Algorithms in C++**. 2nd. ed. Pacific Groove, CA, USA: Thomson Learning, 2001. 664 p.

ELMASRI, R.; NAVATHE, S. **Sistemas de Banco de Dados**. 6a. ed. São Paulo, SP, Brasil: Pearson Addison Wesley, 2011. 788 p.

FAGIN, R.; NIEVERGELT, J.; PIPPENGER, N.; STRONG, H. R. Extendible hashing - a fast access method for dynamic files. **ACM Transactions on Database Systems**, ACM, New York, NY, USA, v. 4, n. 3, p. 315–344, 1979.

FALOUTSOS, C. **Searching Multimedia Databases by Content**. College Park, MD, USA: Kluwer Academic Publishers, 1996. 168 p.

FAYAD, M. E.; SCHMIDT, D. C. Object-oriented application frameworks. **Communications of the ACM**, ACM, New York, NY, USA, v. 40, n. 10, p. 32–38, out. 1997.

FAYAD, M. E.; SCHMIDT, D. C.; JOHNSON, R. E. **Building application frameworks: object-oriented foundations of framework design**. New York, NY, USA: John Wiley & Sons, Inc., 1999.

FREEMAN, E.; FREEMAN, E.; BATES, B.; SIERRA, K. **Design Patterns**. 1st. ed. Sabastopol, CA, USA: O' Reilly & Associates, Inc., 2004. 678 p. (Head First).

GAEDE, V.; GÜNTHER, O. Multidimensional access methods. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 30, n. 2, p. 170–231, jun. 1998.

GAMMA, E.; HELM, R.; JOHNSON, R.; VLISSIDES, J. **Design patterns: elements of reusable object-oriented software**. Boston, MA, USA: Addison-Wesley, 1995. 364 p.

GARCIA-MOLINA, H.; ULLMAN, J. D.; WIDOM, J. **Database Systems: The Complete Book**. 2nd. ed. Upper Saddle River, NJ, USA: Prentice Hall Press, 2008. 1241 p.

GBDI. **Arboretum**. São Carlos: Grupo de Base de Dados e Imagens / Universidade de São Paulo, 2005. Disponível em: <http://www.gbdi.icmc.usp.br/arboretum/>. Acesso em: 16 de fevereiro de 2012.

GEONET. **GeoNet Country Files**. 2012. Disponível em: <`http://earth-info.nga.mil/gns/html/namefiles.htm`>. Acesso em: 16 de maio de 2012.

GRAEFE, G. Volcano - an extensible and parallel query evaluation system. **IEEE Transactions on Knowledge and Data Engineering**, Piscataway, NJ, USA, v. 6, n. 1, p. 120–135, fev. 1994.

GRAND, M. **Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML**. 2nd. ed. Indianapolis, IN, USA: Wiley Publishing, 2002. 592 p.

GUTTMAN, A. R-trees: A dynamic index structure for spatial searching. **ACM SIGMOD Record**, ACM, New York, NY, USA, v. 14, n. 2, p. 47–57, jun. 1984.

HELLAND, P. Database management system. In: LIU, L.; ÖZSU, M. T. (Ed.). **Encyclopedia of Database Systems**. New York, NY, USA: Springer-Verlag, 2009. p. 714–719.

HELLERSTEIN, J. M.; NAUGHTON, J. F.; PFEFFER, A. Generalized search trees for database systems. In: **Proceedings of 21st International Conference on Very Large Data Bases**. Zurich, Switzerland: Morgan Kaufmann Publishers Inc., 1995. (VLBD ’95), p. 562–573.

HETLAND, M. L. The basic principles of metric indexing. In: COELLO, C. A. C.; DEHURI, S.; GHOSH, S. (Ed.). **Swarm Intelligence for Multi-objective Problems in Data Mining**. 1st. ed. Chennai, India: Springer-Verlag Berlim Heidelberg, 2009, (Studies in Computational Intelligence, v. 242). p. 199–232.

ISO/IEC. Norm 11578, **Information technology - Open Systems Interconnection - Remote Procedure Call (RPC)**. Genève, Switzerland: International Organization for Standardization / International Electrotechnical Commission, 1996.

_____. Norm 9834-8, **Procedures for the operation of OSI Registration Authorities**. Genève, Switzerland: International Organization for Standardization / International Electrotechnical Commission, 2008.

ITU. Recommendation, **OSI networking and system aspects - Naming, Addressing and Registration**. Genève, Switzerland: International Telecommunication Union, set. 2004. (Series X: Data Networks And Open System Communications).

JBOSS. **Hibernate**. JBoss Community, 2012. Disponível em: <`http://www.hibernate.org`>. Acesso em: 11 de fevereiro de 2012.

JOHNSON, R. E.; FOOTE, B. Designing reusable classes. **Journal of Object-Oriented Programming**, v. 1, n. 2, p. 22–35, jun. 1988.

JSR-12. **Java Data Objects Specification**. 2012. Disponível em: <`http://www.jcp.org/en/jsr/detail?id=12`>. Acesso em: 11 de fevereiro de 2012.

JSR-220. **Java Persistence API: Enterprise JavaBeans**. 2012. Disponível em: <www.jcp.org/en/jsr/detail?id=220>. Acesso em: 11 de fevereiro de 2012.

KAMEL, I.; FALOUTSOS, C. Hilbert R-tree: An improved R-tree using fractals. In: **Proceedings of 20th International Conference on Very Large Data Bases**. Santiago, Chile: Morgan Kaufmann Publishers Inc., 1994. (VLDB '94), p. 500–509.

KIENZLE, J.; GÉLINEAU, S. AO challenge - implementing the acid properties for transactional objects. In: **Proceedings of the 5th International Conference on Aspect-Oriented Software Development**. Bonn, Germany: ACM, 2006. (AOSD '06), p. 202–213.

KIENZLE, J.; ROMANOVSKY, A. Framework based on design patterns for providing persistence in object-oriented programming languages. **IEE Proceedings - Software**, v. 149, n. 3, p. 77–85, jun. 2002.

KNUTH, D. E. **The Art of Computer Programming: Sorting and Searching**. 2nd. ed. Reading City, MA, USA: Addison Wesley Longman Publishing Co., Inc., 1998. 780 p.

KRUSKAL, J. B. On the shortest spanning subtree of a graph and the traveling salesman problem. **Proceedings of the American Mathematical Society**, George Banta Co., Inc., Menasha, WI, USA, v. 7, n. 1, p. 48–50, fev. 1956.

LARSON, P. Å. Linear hashing with partial expansions. In: **Proceedings of 6th International Conference on Very Large Data Bases**. Montreal, Canada: IEEE Computer Society Proceedings, 1980. p. 224–232.

LEACH, P.; MEALLING, M.; SALZ, R. **A Universally Unique Identifier (UUID) URN Namespace**. Internet Engineering Task Force - IETF, 2005. RFC 4122. Disponível em: <http://www.ietf.org/rfc/rfc4122.txt>. Acesso em: 25 de maio de 2012.

LEAVITT, N. Will nosql databases live up to their promise? **IEEE Computer**, v. 43, n. 2, p. 12–14, 2010.

LEIST, S.; ZELLNER, G. Evaluation of current architecture frameworks. In: **Proceedings of the 21st ACM Symposium on Applied computing**. Dijon, France: ACM, 2006. (SAC '06), p. 1546–1553.

LEVENSHTEIN, V. I. Binary codes capable of correcting deletions, insertions, and reversals. **Soviet Physics Doklady**, v. 10, n. 8, p. 707–710, 1966. *(tradução inglesa do original, publicado em russo.)*.

LITWIN, W. Linear hashing: A new tool for file and table addressing. In: **Proceedings of 6th International Conference on Very Large Data Bases**. Montreal, Canada: IEEE Computer Society Proceedings, 1980. p. 212–223.

LOKOČ, J. On M-tree variants in metric and non-metric spaces. In: **WDS '08 Proceedings of Contributed Papers**. Prague, Czech Republic: Matfyzpress, 2008. (Part I: Mathematics and Computer Sciences, v. 1), p. 230–234.

LOMET, D. B. Boundex index exponential hashing. **ACM Transactions on Database Systems**, ACM, New York, NY, USA, v. 8, n. 1, p. 136–165, 1983.

MANOLOPOULOS, Y.; NANOPOULOS, A.; PAPADOPOULOS, A. N.; THEODORIDIS, Y. **R-Trees: Theory and Applications**. 1st. ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. 194 p. (Advanced Information and Knowledge Processing).

MATTSSON, M. **Evolution and Composition of Object-Oriented Frameworks**. 224 p. Tese (Doutorado em Engenharia de Software) — University of Karlskrona/Ronneby, Ronneby, Sweden, 2000.

MATTSSON, M.; BOSCH, J. Stability assessment of evolving industrial object-oriented frameworks. **Journal of Software Maintenance**, v. 12, n. 2, p. 79–102, 2000.

METSKER, S. J.; WAKE, W. C. **Design Patterns in Java**. 3rd. ed. Boston, MA, USA: Addison-Wesley, 2008. 478 p.

NAVARRO, G. A guided tour to approximate string matching. **ACM Computing Surveys**, ACM, New York, NY, USA, v. 33, n. 1, p. 31–88, mar. 2001.

OLIVEIRA, A. L.; FERRARI, F. C.; PENTEADO, R. A. D.; CAMARGO, V. V. Investigating framework product lines. In: **Proceedings of the 27th ACM Symposium on Applied Computing**. Riva del Garda, Trento, Italy: ACM, 2012. (SAC '12), p. 1177–1182.

OOMMEN, B. J.; LOKE, R. K. S. Pattern recognition of strings with substitutions, insertions, deletions and generalized transpositions. **Pattern Recognition**, Elsevier Science Publishers B. V., Montreal, Canada, v. 30, n. 5, p. 789–800, maio 1997.

OPENGROUP. DCE: Remote Procedure Call, **Specification C309**. Reading, Berkshire, UK: The Open Group CAE, ago. 1994.

OPENOFFICE. **Repository for OpenOffice Extensions**. 2012. Disponível em: <`http://extensions.openoffice.org/en/dictionaries`>. Acesso em: 16 de maio de 2012.

PATELLA, M. **Similarity Search in Multimedia Databases**. 173 p. Tese (Doutorado em Engenharia Eletrônica e Informática) — Università degli Studi di Bologna, Bologna, Italy, 1999.

RAMAKRISHNAN, R.; GEHRKE, J. **Database Management Systems**. 2nd. ed. USA: McGraw-Hill, 1999. 931 p.

ROUSSOPOULOS, N.; KELLEY, S.; VINCENT, F. Nearest neighbor queries. **ACM SIGMOD Record**, ACM, New York, NY, USA, v. 24, n. 2, p. 71–79, maio 1995.

SAMET, H. Spatial data structures. In: KIM, W. (Ed.). **Modern Database Systems: The Object Model, Interoperability, and Beyond**. Reading, MA, USA: ACM Press and Addison-Wesley, 1995. p. 361–385.

SEBESTA, R. W. **Concepts of Programming Languages**. 5. ed. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001. 698 p.

SELLIS, T. K.; ROUSSOPOULOS, N.; FALOUTSOS, C. The R+-tree: A dynamic index for multi-dimensional objects. In: **Proceedings of 13th International Conference on Very Large Data Bases**. Brighton, England: Morgan Kaufmann Publishers Inc., 1987. (VLDB '87), p. 507–518.

SHAFFER, C. A. **Data Structures and Algorithm Analysis**. 3rd. ed. Blacksburg, VA, USA: Dover Publications, 2012. 613 p.

SKOPAL, T. **Metric Indexing in Information Retrieval**. 160 p. Tese (Doutorado em Ciência da Computação e Matemática Aplicada) — VŠB - Technical University of Ostrava (Faculty of Electrical Engineering and Computer Science), Ostrava-Poruba, Czech Republic, 2004.

SKOPAL, T. Pivoting M-tree: A metric access method for efficient similarity search. In: **Proceedings of Dateso 2004: Annual International Workshop on Databases, Texts, Specifications and Objects**. Budapest, Hungary: CEUR-WS.org, 2004. (CEUR Workshop Proceedings, v. 98), p. 27–37.

SKOPAL, T. **Similarity Search in Multimedia Databases**. 18 p. Tese (Habilitation Thesis) — Charles University in Prague (Faculty of Mathematics and Physics), Prague, Czech Republic, 2006.

SKOPAL, T.; LOKOČ, J. New dynamic construction techniques for M-tree. **Journal of Discrete Algorithms**, Elsevier Science Publishers B. V., Amsterdam, The Netherlands, v. 7, n. 1, p. 62–77, mar. 2009.

SKOPAL, T.; POKORNÝ, J.; KRÁTKÝ, M.; SNÁŠEL, V. Revisiting M-tree building principles. In: **Proceedings of 7th East European Conference on Advances in Databases and Information Systems**. Dresden, Germany: Springer-Verlag Berlin Heidelberg, 2003. (Lecture Notes in Computer Science, v. 2798), p. 148–162.

SKOPAL, T.; POKORNÝ, J.; SNÁŠEL, V. PM-tree: Pivoting metric tree for similarity search in multimedia databases. In: **Proceedings of 8th East European Conference on Advances in Databases and Information Systems**. Budapest, Hungary: Springer-Verlag, 2004. (Lecture Notes in Computer Science, v. 3255), p. 1–16.

STONEBRAKER, M.; SELLIS, T. K.; HANSON, E. N. An analysis of rule indexing implementations in data base systems. In: **Proceedings of 1st International Conference on Expert Database Systems**. Charleston, SC, USA: Benjamin Cummings, 1986. p. 465–476.

THOMSON, A.; ABADI, D. J. The case for determinism in database systems. **Proceedings of VLDB Endowment**, VLDB Endowment, v. 3, n. 1-2, p. 70–80, set. 2010.

TRAINA JR., C.; TRAINA, A. J. M.; FALOUTSOS, C.; SEEGER, B. Fast indexing and visualization of metric data sets using slim-trees. **IEEE Transactions on Knowledge and Data Engineering**, IEEE Educational Activities Department, Piscataway, NJ, USA, v. 14, n. 2, p. 244–260, mar. 2002.

TRAINA JR., C.; TRAINA, A. J. M.; SEEGER, B.; FALOUTSOS, C. Slim-trees: High performance metric trees minimizing overlap between nodes. In: **Proceedings of 7th International Conference on Extending Database Technology**. Konstanz, Germany: Springer-Verlag Co., Inc., 2000. (Lecture Notes in Computer Science, v. 1777), p. 51–65.

TZOURAMANIS, T. History-independence: a fresh look at the case of r-trees. In: **Proceedings of the 27th Symposium on Applied Computing**. Riva del Garda, Italy: ACM, 2012. (SAC '12), p. 7–12.

UNIPROT. **UniProtKB/TrEMBL Fasta Download**. 2012. Disponível em: <`http://www.uniprot.org/downloads`>. Acesso em: 16 de maio de 2012.

WESKE, M.; KUROPKA, D. Flexible persistence framework for object-oriented middleware. **Technical Report: Hasso Plattner Institute for Software Systems Engineering**, Potsdam-Babelsberg, Germany, 2001.

XU, W.; MIRANKER, D. P. A metric model of amino acid substitution. **Bioinformatics**, Oxford University Press, Oxford, UK, v. 20, n. 8, p. 1214–1221, maio 2004.

XU, W.; MIRANKER, D. P.; MAO, R.; WANG, S. Indexing protein sequences in metric spaces. **Technical Report: The University of Texas at Austin**, Austin, TX, USA, p. 1–17, out. 2003.

XXL. **Extensible and Flexible Framework**. 2012. Disponível em: <`http://code.google.com/p/xxl/`>. Acesso em: 16 de maio de 2012.

ZAHN, L.; DINEEN, T.; LEACH, P. **Network computing architecture**. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1990. 224 p.

ZEZULA, P.; AMATO, G.; DOHNAL, V.; BATKO, M. **Similarity Search: The Metric Space Approach**. 1st. ed. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006. 220 p. (Advances in Database Systems, v. 32).

_____. _____. 2007. Free Course Slides. Disponível em: <http://www.nmis.isti.cnr.it/amato/similarity-search-book/slides/>. Acesso em: 11 de fevereiro de 2012.

ZHANG, D.; MANOLOPOULOS, Y.; THEODORIDIS, Y.; TSOTRAS, V. J. Extendible hashing. In: LIU, L.; ÖZSU, M. T. (Ed.). **Encyclopedia of Database Systems**. New York, NY, USA: Springer-Verlag, 2009. p. 1162–1164.

ZHOU, X.; WANG, G.; YU, J. X.; YU, G. M+-tree: A new dynamical multidimensional index for metric spaces. In: **Proceedings of 14th Australasian Database Conference**. Darlinghurst, Australia: Australian Computer Society, Inc., 2003. (ADC '03, v. 17), p. 161–168.

Apêndice A

A Métrica de Xu

As primeiras pesquisas que envolveram o uso de métricas entre *strings* surgiram na década de 1960 com o problema do *casamento aproximado de caracteres* (NAVARRO, 2001). Em sua forma mais comum, este problema é descrito como a tarefa de localizar um determinado padrão em um texto, permitindo um número limitado de "erros" entre o padrão e o texto original. Ainda segundo Navarro (2001), naquela época, as principais motivações para o estudo deste problema vinham de campos como a bioinformática, onde sequências podem ser representadas como textos (por exemplo, `{A, C, G, T}` no DNA); recuperação de textos, onde há o problema de correção de erros ortográficos, e no processamento de sinais. Embora o processamento de sinais não tenha se desenvolvido muito fazendo uso da busca aproximada de *strings*, uma importante contribuição surgida neste campo foi a medida de similaridade conhecida como *distância de Levenshtein* ou *distância de edição*.

A distância de Levenshtein (LEVENSHTEIN, 1966) é uma medida de similaridade entre cadeias de caracteres que retorna o número mínimo de operações atômicas necessárias para *transformar uma string em outra*. Estas operações são formalmente definidas como:

- $ins(s, i, c)$: inserção do caractere c na posição i, na *string* s.
- $rem(s, i)$: remoção do caractere da posição i, na *string* s.
- $sub(s, i, c)$: substituição, na *string* s, do caractere da posição i pelo caractere c.

Sendo a uma *string* e $|a|$ seu comprimento, o cômputo da distância de Levenshtein entre duas *strings* a e b é realizado em uma matriz de distância D (também chamada matriz de alinhamento) de dimensões $m \times n$, onde $m = |a|+1$ e $n = |b|+1$. A Figura A.1(a) exemplifica a matriz de distância formada pelas palavras *"tuesday"* e *"thursday"*.

A primeira linha da matriz de distância (linha 0)(Figura A.1(a)) é inicializada com os valores de penalidade (*gap penalty*) para a remoção de caracteres. Cada célula $D_{0,j}$, com $j = 0..n$,

		[0] ε	[1] T	[2] H	[3] U	[4] R	[5] S	[6] D	[7] A	[8] Y
[0]	ε	0	1	2	3	4	5	6	7	8
[1]	T	1								
[2]	U	2								
[3]	E	3								
[4]	S	4								
[5]	D	5								
[6]	A	6								
[7]	Y	7								

(a)

x_S	x_R
x_I	$D_{i,j}$

(b)

x_T		
	x_S	x_R
	x_I	$D_{i,j}$

(c)

Figura A.1: Matriz da distância de Levenshtein
Fonte: O autor

recebe o valor $j * gap_{rem}$. Para a maioria das aplicações, considera-se o valor de penalidade igual a 1, sendo este o custo de remover um caractere.

De forma análoga, a primeira coluna (coluna 0) é inicializada com os valores de penalidade para a inserção de caracteres. Normalmente, considera-se que o custo de inserir é o mesmo de remover um caractere, sendo, portanto, o *gap penalty* da inserção igual ao da remoção. Assim, cada célula $D_{i,0}$, com $i = 0..m$, recebe o valor $i * gap_{ins}$.

Ainda com relação às primeiras linha e coluna da matriz de distância, $D_{i,0}$ (ou $D_{0,j}$) representa a distância de Levenshtein entre uma *string* de tamanho i (ou j) e ε, onde ε é a *string* vazia. Neste caso, é possível notar claramente que i (respectivamente j) remoções são necessárias na *string* não vazia.

Após a inicialização da matriz de distância, cada célula $D_{i,j}$ será adjacente a três outras células, como mostra a Figura A.1(b). A célula imediatamente superior à $D_{i,j}$ contém x_R, que é a penalidade por remover o caractere da posição (i, j). A célula imediatamente à esquerda contém x_I, que é a penalidade por inserir um caractere na posição (i, j). Por fim, a célula da diagonal superior esquerda contém x_S, que é a penalidade por substituir o caractere da posição (i, j) por outro.

De posse dos valores x_I, x_R, x_S e das *strings* a e b, o valor de cada célula $D_{i,j}$ é dado pela Equação A.1:

$$D_{i,j} = min(x_I + 1, x_R + 1, x_S + c_{Sub}) \tag{A.1}$$

e o custo de substituição c_{Sub} é dado pela Equação A.2:

$$c_{Sub} = \begin{cases} 0, & \text{se } a[i] = b[j] \\ 1, & \text{caso contrário} \end{cases} \tag{A.2}$$

Ao término do cálculo de todos os valores de $D_{i,j}$, a matriz de distância obtida será parecida com a da Figura A.2 e a célula $D_{|a|,|b|}$ (mais inferior à direita) conterá o valor da distância de Levenshtein entre as *strings* a e b processadas. No exemplo da Figura A.2, a distância entre as palavras *"tuesday"* e *"thursday"* é $D_{7,8} = 2$. De fato, removendo de *"thursday"* o caractere "H" e substituindo o caractere "R" pela letra "E", forma-se a palavra *"tuesday"*, totalizando 2 operações atômicas.

		[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]
		ε	T	H	U	R	S	D	A	Y
[0]	ε	0	1	2	3	4	5	6	7	8
[1]	T	1	0	1	2	3	4	5	6	7
[2]	U	2	1	1	1	2	3	4	5	6
[3]	E	3	2	2	2	2	3	4	5	6
[4]	S	4	3	3	3	3	2	3	4	5
[5]	D	5	4	4	4	4	3	2	3	4
[6]	A	6	5	5	5	5	4	3	2	3
[7]	Y	7	6	6	6	6	5	4	3	2

Figura A.2: Cálculo da distância de Levenshtein entre *"tuesday"* e *"thursday"*
Fonte: O autor

A distância de Levenshtein possui complexidade de tempo proporcional a $\mathcal{O}(mn)$, devido aos laços necessários para processar as células das m linhas e n colunas da matriz D. A complexidade espacial da distância de Levenshtein também é igual a $\mathcal{O}(mn)$, que representa o espaço necessário para acomodar a matriz D na memória (CORMEN et al., 2009).

O trabalho de Damerau (1964) introduziu outra métrica entre *strings*, que, com o tempo, passou a ser chamada de *distância de Damerau-Levenshtein*. A distância de Damerau-Levenshtein utiliza as mesmas 3 operações atômicas da distância de Levenshtein em adição com a operação de *transposição*. A operação de transposição é formalmente definida como $trans(s, ab, ba)$, o que significa que, na *string* s, os dois caracteres ab adjacentes serão invertidos, trocados de posição, tornando-se ba.

Para o cálculo da distância de Damerau-Levenshtein, a matriz de distância é inicializada de maneira idêntica ao cálculo da distância de Levenshtein. Entretanto, como mostra a Figura A.1(c), uma célula $D_{i,j}$ terá uma nova célula "vizinha", igual a x_T. O valor x_T representa a penalidade associada à operação de transposição. Pela posição da célula de valor x_T, a transposição é calculada apenas para as células em que $i > 1$ e $j > 1$, ou seja, a partir das terceiras linha e coluna da matriz.

O preenchimento da matriz de distância, segundo a métrica de Damerau-Levenshtein, é dado pela Equação A.3:

$$D_{i,j} = min(x_I + 1, x_R + 1, x_S + c_{Sub}, x_T + c_{Trans}) \tag{A.3}$$

O custo de substituição c_{Sub} é o mesmo apresentado pela Equação A.2 e o recém adicionado custo de transposição c_{Trans} é dado de acordo com a Equação A.4:

$$c_{Trans} = \begin{cases} 0, & \text{se } a[i] = b[j-1] \wedge a[i-1] = b[j] \\ 1, & \text{caso contrário} \end{cases} \tag{A.4}$$

As complexidades de tempo e espaço da distância de Damerau-Levenshtein permanecem idênticas as da distância de Levenshtein ($\mathcal{O}(mn)$), pois apenas a operação de transposição foi adicionada. O Algoritmo A.1 (OOMMEN; LOKE, 1997), que faz uso de programação dinâmica, apresenta o cálculo da distância de Damerau-Levenshtein.

O Algoritmo A.1 também pode ser usado para computar a distância de Levenshtein, bastando remover o condicional das linhas 20−22, ou seja, o cálculo da transposição.

Algoritmo A.1: Damerau-Levenshtein

Entrada: As cadeias de caracteres $a[\,]$ e $b[\,]$.
Dados: Os *gap penalties* de inserção e remoção.
Saída: A distância entre as cadeias a e b.

```
1  m ← |a| + 1; // onde: |a| = comprimento de a[ ]
2  n ← |b| + 1;
3  Declarar a matriz d[0..m, 0..n];
4  para i ← 0 até m faça
5      d[i, 0] ← i * gap_ins;
6  para j ← 1 até n faça
7      d[0, j] ← j * gap_rem;
8  // Assumindo que o primeiro índice da string é 0.
9  para i ← 1 até m faça
10     para j ← 1 até n faça
11         se a[i − 1] = b[j − 1] então
12             custo ← 0;
13         senão
14             custo ← 1;
15         // Cálculo das penalizações
16         ins ← d[i, j − 1] + 1; // inserção
17         rem ← d[i − 1, j] + 1; // remoção
18         sub ← d[i − 1, j − 1] + custo; // substituição
19         d[i, j] ← mínimo(ins, rem, sub);
20         se i > 1 ∧ j > 1 ∧ a[i − 1] = b[j − 2] ∧ a[i − 2] = b[j − 1] então
21             trans ← d[i − 2, j − 2] + custo ; // transposição
22             d[i, j] ← mínimo(d[i, j], trans);
23 retorna d[m − 1, n − 1];
```

Ainda com relação ao Algoritmo A.1, os valores de c_{Sub} e c_{Trans} são combinados na variável *custo*, visto que ambos assumem apenas o valor 0, se os caracteres comparados são iguais, ou 1, caso contrário. Entretanto, em certos domínios de aplicação, os valores 0 e 1 podem não condizer com o custo real em substituir ou transpor um caractere, como ocorre com a base de proteínas dos experimentos do Capítulo 5.

Na base de dados UniProt (Seção 5.1), as proteínas são representadas através de *strings* de tamanhos distintos. Cada caractere de uma *string* representa um dos 20 aminoácidos existentes, formadores da proteína. Devido a estrutura química de um aminoácido, a qual não será detalhada neste trabalho, o custo de substituição / transposição de um aminoácido por outro é variável e dependente dos aminoácidos envolvidos na operação. Isto significa que o cômputo da distância de Damerau-Levenshtein entre duas cadeias de caracteres, representativas de proteínas, *não* é uma medida real do valor de similaridade entre essas proteínas.

Com base neste problema, os trabalhos de Xu et al. (2003) e Xu e Miranker (2004) introduziram uma métrica entre proteínas representadas por cadeias de caracteres, a qual é referida neste trabalho como *métrica de Xu*. A métrica de Xu consiste de uma variação da distância de Damerau-Levenshtein onde o custo da substituição / transposição de um aminoácido por outro é dado de acordo com uma matriz chamada de mPAM (*Matrix Point Accepted Mutation*) (XU; MIRANKER, 2004).

Tabela A.1: Matriz mPAM

	A	R	N	D	C	Q	E	G	H	I	L	K	M	F	P	S	T	W	Y	V
A	0	2	2	2	3	2	2	2	2	2	2	2	2	3	2	2	2	5	4	2
R	2	0	2	2	4	2	2	2	2	3	3	2	2	4	2	2	2	4	4	3
N	2	2	0	2	4	2	2	2	2	3	3	2	2	4	2	2	2	5	4	2
D	2	2	2	0	4	2	2	2	2	3	3	2	3	4	2	2	2	6	4	2
C	3	4	4	4	0	4	4	3	4	3	4	4	4	4	3	3	3	7	3	3
Q	2	2	2	2	4	0	2	2	2	3	3	2	2	4	2	2	2	5	4	3
E	2	2	2	2	4	2	0	2	2	3	3	2	3	4	2	2	2	6	4	2
G	2	2	2	2	3	2	2	0	2	2	3	2	2	4	2	2	2	6	4	2
H	2	2	2	2	4	2	2	2	0	3	3	2	3	3	2	2	2	5	3	3
I	2	3	3	3	3	3	3	2	3	0	1	3	2	2	2	2	2	5	3	2
L	2	3	3	3	4	3	3	3	3	1	0	3	1	2	3	3	2	4	2	1
K	2	2	2	2	4	2	2	2	2	3	3	0	2	4	2	2	2	4	4	3
M	2	2	2	3	4	2	3	2	3	2	1	2	0	2	2	2	2	4	3	2
F	3	4	4	4	4	4	4	4	3	2	2	4	2	0	4	3	3	3	1	2
P	2	2	2	2	3	2	2	2	2	2	3	2	2	4	0	2	2	5	4	2
S	2	2	2	2	3	2	2	2	2	2	3	2	2	3	2	0	2	5	4	2
T	2	2	2	2	3	2	2	2	2	2	2	2	2	3	2	2	0	5	3	2
W	5	4	5	6	7	5	6	6	5	5	4	4	4	3	5	5	5	0	4	5
Y	4	4	4	4	3	4	4	4	3	3	2	4	3	1	4	4	3	4	0	3
V	2	3	2	2	3	3	2	2	3	2	1	3	2	2	2	2	2	5	3	0

Fonte: Xu e Miranker (2004)

A matriz mPAM, apresentada pela Tabela A.1, é uma matriz métrica de substituição de aminoácidos, que integra tendências evolucionárias em relação a métodos anteriormente empregados. O estudo de Xu e Miranker (2004) baseou-se no cálculo de tempo esperado entre as substituições. Assim, um par de aminoácidos com uma alta taxa de substituição deve levar menos tempo para aparecer em uma sequência proteica do que um par com uma taxa de substituição menor. Consequentemente, os pares de aminoácidos cuja sequência é mais similar têm um custo mais próximo de zero, o que é uma exigência da métrica de Xu.

Com base na matriz mPAM, para calcular a métrica de Xu, basta alterar a variável *custo* do Algoritmo A.1 (linhas 11−14) para receber o valor expresso na mPAM. Este valor é encontrado na matriz de acordo com os caracteres $a[i-1]$ e $b[j-1]$ (linha 11 do Algoritmo A.1), onde $a[i-1]$ está disposto nas linhas da mPAM e $b[j-1]$ nas colunas.

O trabalho de Xu et al. (2003) realiza a validação da métrica de Xu. A partir dos experimentos realizados, os autores mostram que a mesma é útil na indexação de proteínas, no espaço métrico, com eficiência e escalabilidade. Já em Xu e Miranker (2004), a matriz mPAM é formalmente apresentada e comparada com métodos anteriormente utilizados na substituição de aminoácidos (PAM, PAM250 etc). Este trabalho ainda mostra que a precisão da mPAM supera a dos outros métodos discutidos.

Anotações