UNIVERSIDADE FEDERAL FLUMINENSE

NILSON LUÍS DAMASCENO

Tinycubes: Tecnologia Modular para Exploração Visual e Interativa de Grandes Volumes de Dados Espaço-temporais Multidimensionais Gerados Continuamente

NITERÓI

2020

UNIVERSIDADE FEDERAL FLUMINENSE

NILSON LUÍS DAMASCENO

Tinycubes: Tecnologia Modular para Exploração Visual e Interativa de Grandes Volumes de Dados Espaço-temporais Multidimensionais Gerados Continuamente

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Sistemas de Computação

Orientador:
Antônio Augusto de Aragão Rocha

NITERÓI

2020

Ficha catalográfica automática - SDC/BEE
Gerada com informações fornecidas pelo autor

D155t Damasceno, Nilson Luís
Tinycubes: Tecnologia Modular para Exploração Visual e Interativa de Grandes Volumes de Dados Espaço-temporais Multidimensionais Gerados Continuamente / Nilson Luís Damasceno ; Antonio Augusto de Aragão Rocha, orientador. Niterói, 2020.
179 f. : il.

Dissertação (mestrado)-Universidade Federal Fluminense, Niterói, 2020.

DOI: http://dx.doi.org/10.22409/PGC.2020.m.88242200700

1. Data Analysis. 2. Visual Exploration. 3. Datacubes. 4. Spatio-temporal. 5. Produção intelectual. I. Rocha, Antonio Augusto de Aragão, orientador. II. Universidade Federal Fluminense. Instituto de Computação. III. Título.

CDD -

Bibliotecário responsável: Sandra Lopes Coelho - CRB7/3389

NILSON LUÍS DAMASCENO

Tinycubes: Tecnologia Modular para Exploração Visual e Interativa de Grandes Volumes de Dados Espaço-temporais Multidimensionais Gerados Continuamente

Dissertação de Mestrado apresentada ao Programa de Pós-Graduação em Computação da Universidade Federal Fluminense como requisito parcial para a obtenção do Grau de Mestre em Computação. Área de concentração: Sistemas de Computação.

Aprovada em março de 2020.

Banca Examinadora:

Prof. Antonio Augusto de Aragão Rocha, UFF – Presidente

Prof. Marcos de Oliveira Lage Ferreira, UFF

Prof. Artur Ziviani, LNCC

Niterói

2020

Dedico este trabalho à minha tia e madrinha,
Nancy Damasceno (in memorian),
que sempre me apoiou.

Agradecimentos

Ao criador, que nos permite vivenciar tantas experiências estimulantes.

À minha amiga e companheira Moema, por todo o apoio dado durante esta jornada.

Ao meu caríssimo colega, Rogério Ferreira de Moraes, sempre gentil e disposto a ajudar com seus conhecimentos matemáticos e comentários inteligentes.

Ao meu amigo e colega, Leonardo Pio Vasconcelos, que ao dedicar horas do seu tempo revisando e comentando o texto deste trabalho, acabou se transformando, para mim, em um "sub-orientador".

Ao meu colega, Lucas Diego Barbosa, que prestou uma ajuda inestimável na reta final do trabalho.

Ao meu Orientador Antônio Augusto de Aragão Rocha, por me oferecer um trabalho tão rico e desafiador.

A todos os meus familiares e amigos pelo apoio e colaboração.

Resumo

A análise de dados é uma atividade essencial para cientistas, gestores públicos e privados, além de outros tomadores de decisão. A evolução da tecnologia tornou possível obter informações analíticas elaboradas através da exploração visual e interativa de grandes volumes de dados complexos, compostos por múltiplos atributos, incluindo coordenadas espaciais e séries temporais. Neste contexto, a estrutura de dados Nanocubes, inspirada na tecnologia *OnLine Analytical Processing* (OLAP), destaca-se pela eficiência demonstrada ao operar centenas de milhões de dados complexos utilizando computadores de fácil aquisição, enquanto consegue apresentar tempos de latência adequados para manter a interatividade com usuários. No entanto, essa tecnologia possui algumas limitações, como a impossibilidade de receber dados após ter produzido informações, a total incapacidade de remover dados recebidos e a ausência de facilidades para lidar com novos tipos de dados e novos tipos de informação, bem como novos métodos para extração de informações. Além disso, os estudos teóricos que realizamos detectaram características do Nanocubes que comprometem diretamente a eficiência no uso da memória e, indiretamente, no processamento, impedindo a utilização de equipamentos mais econômicos.

Este trabalho apresenta a estrutura de dados Tinycubes, que utiliza algoritmos que oferecem, ao mesmo tempo, um aumento de eficiência de até 24% no consumo memória e uma ligeira redução no tempo de processamento em relação ao Nanocubes, considerado o estado-da-arte em seu segmento. A estrutura também apresenta novos algoritmos, capazes de inserir e remover dados durante a operação. A nova capacidade de remover dados possibilita a reciclagem do uso da memória, permitindo que o Tinycubes funcione como um *In-memory OLAP Server*, continuamente recebendo dados e disponibilizando, em tempo real, informações extraídas deles. No aspecto teórico, o trabalho apresenta as análises de eficiência realizadas, bem como a especificação formal da estrutura Tinycubes utilizando a Teoria dos Grafos. No aspecto operacional, o trabalho introduz um conjunto inédito de recursos para o seu segmento, dos quais se destacam: um sistema de módulos flexível, que possibilita a instalação de novos tipos de dados, novos tipos de informação e também de técnicas alternativas para a extração de informação; uma linguagem de consulta para exploração de informações inspirada na linguagem SQL e baseada em JSON; um processador de consultas agnóstico, capaz de operar, sem viés, diferentes tipos de dados e informações.

Em resumo, a tecnologia Tinycubes, formada por todos esses componentes, é uma solução tecnológica econômica e eficiente, capaz de superar todas as dificuldades e limitações identificadas na tecnologia considerada o estado-da-arte.

Palavras-chave: VIS, data analysis, Big Data, visual exploration, spatio-temporal, datacubes.

Abstract

Data analysis is an essential activity for scientists, public and private managers and other decision makers. The evolution of technology has made it possible to extract hight quality analytical information using visual and iterative exploration of large volumes complex data, composed by multiple attributes, including spatial coordinates and time series. In this context, the Nanocubes data structure, inspired by OLAP technology, stands out for its demonstrated efficiency by operating hundreds of millions of complex data using easily commodities computers, while offering adequate latency times for user interaction. However, this technology has some limitations, such as the impossibility receiving data after any information production, the complete inability to remove data received and the lack of facilities to deal with new types of data and new types information, as well as new methods for extracting information. In addition, theoretical studies that we carried out identified characteristics of the nanocubes structure that directly compromise the science and use of memory, and indirectly the ability process, reducing their ability to use more economical equipment.

This work presents the Tinycubes data structure, which uses algorithms that offer, at same time, up to 24% increase in memory efficiency and a slight reduction in processing time compared to Nanocubes, considered the state-of-the-art in your segment. The structure has also new algorithms capable of inserting and removing data during operation. This ability to remove data allows recycling of memory, enabling Tinycubes to operate as an In-memory OLAP Server, continuously receiving data and making available, in real time, information extracted from them. In the theoretical side, the work presents the analysis of efficiency carried out, and the formal specification of the Tinycubes structure using Graph Theory. In the operational side, the work introduces an unprecedented set of resources for its segment, with the following highlights: a flexible module system, which allows the installation of new types of data, new types of information and alternative techniques for extracting information; a query language for information exploration inspired by SQL language and based on JSON; an agnostic query processor, capable of operating, without bias, different types of data and information.

In summary, Tinycubes technology, made up of all these components, is an economical and efficient technological solution, capable of overcoming all difficulties and limitations identified in the technology considered state-of-the-art.

Keywords: VIS, Data Analysis, OLAP, Big Data, Visual Exploration, spatio-temporal, datacubes.

Lista de Figuras

2.1 Datacubes bidimensionais 8

2.2 Três exemplos de dimensões hierárquicas 10

2.3 Representação gráfica para grafos 16

2.4 Representação gráfica para árvores 19

2.5 Quadtree indexando 5 subáreas com precisão de 1/16 da área total 21

4.1 Datacubes bidimensionais 31

4.2 Representação didática de um nanocube para tabela na Figura 4.1(b) . . . 32

4.3 Nanocube alternativo para a tabela na Figura 4.1(b) 33

4.4 Datacube apresentando o tempo como uma hierárquia 33

4.5 Nanocube com hierarquia de tempo para datacube da Figura 4.4 34

4.6 Datacube da Figura 4.4 na ordem "produto" → "ano / semestre" 35

4.7 Representação do nanocube da Figura 4.3 (quase) no formato final 36

4.8 Exemplo de uso de aresta Shared Content 38

4.9 Exemplo de uso de aresta Shared Child 39

4.10 Algoritmo de inserção do Nanocubes Fonte: Nanocubes [38] 40

4.11 Nanocube identificando vértices de ligação 41

4.12 Caminhos entre dimensões em um nanocube 42

4.13 Instância da estrutura de dados proposta equivalente ao nanocube anterior 42

4.14 Nanocube após algumas inserções 43

4.15 Etapas da inserção do par (7,9) 45

4.16 Resultado final e desejado após inserção do par (7,9) 45

4.17 Etapas da inserção do par (7,8) 46

4.18 Resultado final e desejado após inserção do par (7,8) 46

5.1 Esquema simplificado da Tecnologia Tinycubes 48

5.2 Índice e Terminais 50

5.3 Diferenças em relação a uma árvore de busca 51

5.4 Exemplo de Record 52

6.1 Representação gráfica para vértices de um Tinycubes 68

6.2 Representação gráfica para as de arestas usadas em um Tinycubes 70

6.3 Fragmentos de tinycubes com L=2 mas diferentes valores de D 72

6.4 Tinycubes completos com L=2 mas diferentes valores de D 73

6.5 Exemplo de tinytrees 78

6.6 Maxitree e subtrees 80

7.1 Oportunidades de modularização 89

7.2 Diagrama esquemático do sistema de módulos 90

8.1 Schema descrito na Listagem 8.1 107

9.1 Representação visual de tinycube produzida pelo programa “dot” 116

9.2 Consequências da peculiaridade 1 118

9.3 Consequências da peculiaridade 2 118

9.4 Exemplo de superação da peculiaridade 1 do Nanocubes 120

9.5 Exemplo de superação da peculiaridade 2 do Nanocubes 120

9.6 Criação de SHARED CHILD para compartilhamento 121

9.7 Criação de mais uma tinytree sem apresentar a peculiaridade 1 122

9.8 Diversas reinserções sem ocorrência da peculiaridade 2 122

9.9 Bifurcação na dimensão 2 123

9.10 Consequências de duas remoções que zeraram o CI de um vértice terminal 124

9.11 Remoçao tripla afetando apenas CIs 125

9.12 Remoções eliminando SHAREDs CHILD e tinytrees 126

9.13 Schema Brightkite 128

9.14 Brightkite - melhoria no consumo de memória 128

9.15 Brightkite - Vértices terminais não compartilhados 129

9.16 Brightkite - Vértices terminais compartilhados 131

9.17 Brightkite - total de vértices 131

9.18 Brightkite - Tempo de carga 132

9.19 Brightkite - valores absolutos 133

9.20 Heatmap baseado no dataset CelularTaxisRio 134

9.21 Schema CelularTaxisRio 134

9.22 CelularTaxisRio - melhoria no consumo de memória 135

9.23 CelularTaxisRio - valores absolutos 135

9.24 CelularTaxisRio - Vértices Terminais 136

9.25 CelularTaxisRio - total de vértices 137

9.26 CelularTaxisRio - Tempo de carga 138

9.27 Visão esquemática da ferramenta Network Borescope 140

9.28 Network Borescope 143

9.29 Detalhes da interface incluindo os métodos IA 144

9.30 Network Borescope - Consultas simultâneas 146

9.31 Network Borescope - Detecção de Anomalias 147

Lista de Tabelas

6.1 Total de pacotes e Total de bytes recebidos 64

6.2 Tamanho médio do pacote e Maior quantidade de bytes 64

9.1 Alguns dos cenários de teste de estresse 127

Lista of Algoritmos

6.1 Algoritmo de inserção para o Tinycubes 81

6.2 Inserção de um record numa tinytree 81

6.3 Algoritmo de descida para inserção de um record 82

6.4 Algoritmo de subida durante a inserção de um record 84

6.5 Algoritmo de remoção para o Tinycubes 86

6.6 Remoção numa tinytree 86

6.7 Remoção - descida na tinytree 87

6.8 Remoção - retorno da subida ajustando a tinytree 88

Relação de Listagens

7.1 Registro da classe geo 93

7.2 Registro das operações para classe geo 94

7.3 Content Avg (average - media) 95

7.4 Registro do Content Avg 96

7.5 Registro do Conteiner binlist 97

7.6 Registro de binlist como Modulo Dimensional 97

7.7 Registro de binlist como Modulo Content 98

7.8 Registro da operação between 98

8.1 Fragmento de um arquivo schema 100

8.2 Exemplo de consulta 103

8.3 BNF da consulta 104

8.4 BNF da resposta 105

8.5 Consulta elementar 108

8.6 Consulta básica 108

8.7 Exemplo de consulta com agregação de Contents 109

8.8 Exemplo de consulta com restrição de tempo 109

8.9 Exemplo de consulta temporal 110

8.10 Exemplo de consulta espacial 110

8.11 Exemplo de group-by básico 111

8.12 Exemplo de group-by com agregação de Contents 111

8.13 Exemplo de group-by sobre Container 112

8.14 Exemplo de group-by sobre Container 112

9.1 Limites de exibição 142

9.2 Exemplo de consultas enviadas 143

9.3 Respostas (parte) para as consultas realizadas 145

Lista de Abreviaturas e Siglas

OLAP *OnLine Analytical Processing* iv

UFF *Universidade Federal Fluminense*

RNP *Rede Nacional de Ensino e Pesquisa*

IA *Inteligência Artificial*

Sumário

1 Introdução **1**

1.1 Requisitos para tecnologia 2

1.2 Questões e Atividades de Pesquisa 4

1.3 Objetivo 6

1.4 Organização deste documento 6

2 Prolegômenos **7**

2.1 OLAP, Datacubes, Dimensões e Hierarquias 7

2.2 Exploração visual e interativa de dados - EVID 11

2.2.1 Requisitos para EVID 12

2.2.2 Alguns desafios 13

2.3 Record como conjunto 14

2.4 Teoria dos Grafos 14

2.5 Grafos e Árvores como estruturas de dados 18

2.6 Tiles - visualização de mapas 20

2.7 Quadtrees 21

3 Trabalhos Relacionados **23**

3.1 Gerenciamento de dados para EVID 23

3.2 Nanocubes 25

3.3 Considerações finais 28

4 Nanocubes - Análises de eficiência **30**

4.1 Introdução ao Nanocubes 30
4.1.1 Visão geral 30
4.1.2 Conceitos, definições e propriedades 35
4.2 Nanocubes - Organização e regras relevantes 40
4.3 Análise da estrutura Nanocubes 41
4.3.1 Análise 41
4.3.2 Estrutura alternativa 42
4.3.3 Considerações 43
4.4 Análise do algoritmo de inserção 43
4.4.1 Análise 44
4.4.1.1 Peculiaridade 1: Valores inéditos 44
4.4.1.2 Peculiaridade 2: Reinserção 45
4.4.2 Considerações 47
4.5 Conclusão 47

5 Tinycubes - Visão geral 48

5.1 Introdução 48
5.2 Conceitos essenciais 49
5.3 Estrutura de dados 50
5.3.1 Índice e terminais 50
5.4 Conclusão 53

6 Tinycubes - Estrutura de Dados 54

6.1 Schema e Grafo dimensional 54
6.1.1 Schema 54
6.1.2 Grafo dimensional - G 55
6.2 Elementos do Schema 56

6.2.1 Record 56
6.2.2 Funções mapeadoras 56
6.2.3 Contents 57
6.2.3.1 Content 58
6.2.3.2 Inserção em um content 58
6.2.3.3 Extração de informação de um content 59
6.2.3.4 Relevância da ordem de inserção de records 59
6.2.3.5 Remoção de dados de um content 60
6.2.3.7 Agregação de múltiplos contents 62
6.2.4 Containers 65
6.2.5 SContent 66
6.3 Elementos do Grafo Dimensional 66
6.3.1 Definições e propriedades iniciais 66
6.3.2 Vértices dimensionais 67
6.3.3 Arestas dimensionais 68
6.3.4 Características estruturantes 71
6.3.5 Trees: Tinytrees, Maxitree, ... 73
6.3.6 Propriedades construtivas fundamentais 74
6.3.6.1 Arestas CUBE 74
6.3.6.2 Arestas CHILD 75
6.3.7 Exemplo 78
6.4 Algoritmos de Inserção e de Remoção 79
6.4.1 Algoritmo de Inserção 80
6.4.2 Algoritmo de Remoção 84

7 Tinycubes - Modularização 89

7.1 Introdução 90

7.2 Módulos Dimensionais 91
7.3 Módulos Content 95
7.4 Módulos Container 97
7.5 Módulos Estruturais 98

8 Tinycubes - Tecnologia 100

8.1 Arquitetura Cliente x Servidor 100
8.2 Especificação do Schema 100
8.3 Linguagem de consulta 103
8.3.1 Consulta 103
8.3.2 Comandos 104
8.3.3 Resposta 105
8.4 Processador de consultas agnóstico - AQP 106
8.4.1 Consultas sem "group-by" (consulta simples) 108
8.4.2 Consultas com "group-by" 111
8.4.3 Processador de Consultas Agnóstico 113

9 Resultados e Caso de uso 115

9.1 Configurações e programas utilizados 115
9.2 Resultados com datasets de validação 116
9.2.1 Introdução 116
9.2.2 Sequência de Inserções - Nanocubes+ 117
9.2.3 Sequência de Inserções 120
9.2.4 Sequência de Remoções 123
9.3 Resultados com datasets sintéticos 124
9.4 Resultados com datasets reais 126
9.4.1 Brightkite 128

9.4.2 Operadoras de telefonia utilizadas por táxis 133

9.4.3 Conclusão 139

9.5 Caso de uso - Ferramenta Network Borescope 140

9.5.1 Introdução 140

9.5.2 Dados e Schema 141

9.5.3 Interface web ↔ Servidor Tinycubes 142

10 Conclusão **148**

10.1 Considerações finais 148

10.2 Contribuições 148

10.3 Limitações e Trabalhos Futuros 150

Referências **153**

Capítulo 1

Introdução

A demanda por ferramentas que auxiliam na análise de dados não é recente [69]. O desenvolvimento de técnicas de análise estatística e de visualização de dados é uma consequência direta dessa demanda. Com a evolução da tecnologia, a capacidade dessas técnicas aumentou consideravelmente [78, 18]. Porém a mesma evolução tecnológica que tornou possível uma maior capacidade operacional, também trouxe novos desafios ao aumentar o volume e a complexidade dos dados a serem analisados. Os desafios mais diretos decorrem do fato de que, quanto maior o volume e a complexidade dos dados, maior tende a ser o tempo de resposta à solicitação de informações, o que pode tornar a interatividade impossível.

Existem diversas abordagens para viabilizar a exploração visual e interativa de grandes volumes de dados complexos [41, 7, 34]. Em algumas delas, abre-se mão da precisão da informação gerada para garantir um tempo de resposta reduzido [12, 54, 8]. Em outros casos, tenta-se antever possíveis consultas e deixar preparadas respostas visuais prontas, como no caso do imMens [43]. Outra abordagem é memorizar partes de respostas complexas para que consultas subsequentes que reutilizem essas partes, tenham o tempo total de resposta reduzido [52]. Ao mesmo tempo, a evolução na comunicação de dados, permitiu que algumas técnicas utilizassem conjuntos de computadores para realizar a extração de informação mais rapidamente [52]. Com o desenvolvimento das GPUs, outras técnicas criaram algoritmos específicos para explorar o paralelismo maciço oferecido [26, 79, 33, 78]. Finalmente, algumas técnicas almejam superar o desafio de manter o tempo de resposta baixo, investindo em algoritmos especializados e altamente eficientes que são capazes de operar com grandes volumes de dados com baixo tempo de resposta utilizando apenas um computador convencional [38]. Este trabalho segue a linha definida por essa última abordagem.

O problema de realizar análises computacionais sobre grandes volumes de dados através de estruturas de dados especializadas tem sido abordado de várias formas [37, 20, 50, 68]. Uma abordagem que vem apresentando bons resultados é baseada na utilização de datacubes [36, 55, 14], estruturas de dados logicamente organizadas como arrays multidimensionais onde cada célula armazena informações pré-computadas. Os datacubes são utilizados em ferramentas OLAP [9] dedicadas ou oferecidos como rotinas auxiliares em programas comuns, como planilhas eletrônicas [15].

A estrutura de dados Nanocubes [38] foi desenvolvida à partir dos conceitos e técnicas já existentes para tecnologia OLAP e para datacubes (Seção 2.1). Essa estrutura é considerada, em trabalhos recentes [39, 35], o estado da arte na utilização de datacubes aplicados à exploração visual e interativa de grandes volumes de dados. Como veremos mais adiante (Capítulo 3), a estrutura Nanocubes atende a grande parte dos requisitos listados na seção 1.1. Porém, ela não pode ser considerada uma solução completa, uma vez que existem algumas limitações importantes com respeito a essa tecnologia, como por exemplo, sua inabilidade de receber mais dados após o início da produção de informações.

1.1 Requisitos para tecnologia

Para que se possa realizar uma avaliação sistemática das capacidades das soluções tecnológicas, é identificar características relevantes[1] encontradas em ferramentas que, através da exploração visual e interativa de grandes volumes de dados complexos (multidimensionais, eventualmente contendo coordenadas espaciais e armazenáveis em longas séries temporais), permitem a extração de informações para realização de análise de dados. Como veremos mais adiante, essas características deverão servir de base para elaboração de um solução abrangente para exploração visual e interativa de dados e, por conta disso, são consideradas requisitos para tecnologia.

Esta seção apresenta requisitos para a tecnologia desejada, nos quais cada característica relevante é relacionada individualmente. Dado que a lista é relativamente extensa, os requisitos foram agrupados em categorias para facilitar o entendimento. A seguir segue a lista dos requisitos que uma tecnologia deve atender[2].

[1] segundo nosso entendimento.

[2] A palavra "armazenar" foi intencionalmente evitada na relação de requisitos porque, apesar da tecnologia realizar armazenamento, o tipo de "valor" a ser armazenado pode não ser nem um dado e nem uma informação útil. Utilizar dado ou informação como sendo o objeto de armazenamento geraria um falso requisito. Por outro lado, definir um termo novo na lista de requisitos para servir como objeto de armazenamento introduziria uma complexidade desnecessária para o momento.

Requisitos Essenciais

Req1 **Transformar dados em informação** - Operar[3] com dados, recebendo-os e posteriormente respondendo a consultas externas que solicitem informações a serem obtidas à partir deles;

Req2 **Produzir informações precisas** - Produzir respostas contendo informações precisas e completas[4];

Req3 **Operar grandes volumes de dados** - Ser capaz de operar com grande volume de dados, na casa de milhões de dados ou mais;

Req4 **Ter baixa latência** - responder às consultas em tempo reduzido, viabilizando a interatividade;

Requisitos sobre tipos de dados

Req5 **Operar dados multidimensionais** - Operar com dados contendo múltiplos atributos, inclusive relacionados a locais e a épocas, sendo eventualmente compostos por outros dados de forma hierárquica;

Req6 **Operar coordenadas espaciais** - Operar com dados que permitam à extração eficiente de informações à partir de coordenadas espaciais (como latitude e longitude) com resolução configurável;

Req7 **Operar séries temporais** - Operar eficientemente com dados armazenados em séries temporais indexadas em qualquer precisão, como anos, dias, segundos, etc.;

Requisitos sobre operação de dados

Req8 **Receber dados adicionais** - ser capaz de receber novos dados, mesmo após respostas já terem sido geradas[5];

Req9 **Receber dados gerados continuamente** - ser capaz de receber dados gerados continuamente, oferecendo algum mecanismo contra esgotamento da memória disponível;

Req10 **Descartar dados irrelevantes** - oferecer algum mecanismo para descarte de valores armazenados que se tornaram irrelevantes durante a exploração;

[3] O termo "operar" é intencionalmente vago. Este termo é usado com o objetivo de postergar uma definição precisa das ações a que serão realizadas sobre os dados.

[4] em contraposição a informações aproximadas ou geradas progressivamente, como ocorre em algumas tecnologias, e que é aceitável apenas em alguns cenários específicos de uso

[5] em contraposição à sistemas "add-only" que não permitem a inserção de novos dados após começar a responder.

Requisitos para evolutibilidade

Req11 **Aceitar novos tipos de dados** - permitir que novos tipos de dados coletados possam ser facilmente utilizados;

Req12 **Aceitar novos tipos de informação** - permitir que novos tipos de informação, bem como novos métodos para extração de informação possam ser facilmente utilizados;

Requisitos operacionais

Req13 **Ser facilmente utilizável** - oferecer uma forma padronizada para o recebimento de consultas e devolução de respostas, possibilitando seu uso por diferentes sistemas de exploração visual e interativa;

Req14 **Ser eficiente** - ser eficiente na utilização dos recursos necessários para operações com os dados;

Req15 **Ser acessível** - ser capaz de operar os dados utilizando computadores de uso cotidiano, incluindo notebooks.

1.2 Questões e Atividades de Pesquisa

A incapacidade da tecnologia Nanocubes, considerada o estado da arte, em atender alguns dos requisitos relacionados (como será visto em detalhes no capítulo 3), levanta questões referentes à possibilidade de se criar uma solução que consiga, de fato, atender simultaneamente a todos os requisitos listados na Seção 1.1. Ao mesmo tempo, para responder à cada uma das questões de pesquisa levantadas, é necessário executar de uma ou mais atividades que respondem a essas questões. A seguir, listam-se as questões de pesquisa identificadas, bem como as atividades a serem realizadas para responder a cada uma dessas questões.

QP1 **Existe alguma forma de melhorar a eficiência da estrutura de dados considerada o estado da arte**?

AP1 Analisar a organização da estrutura de dados considerada o estado da arte visando encontrar oportunidades para aumentar a eficiência na utilização de recursos;

AP2 Analisar os algoritmos utilizados pela estrutura de dados considerada o estado da arte visando encontrar oportunidades para aumentar a eficiência na utilização de recursos;

AP3 Desenvolver e especificar formalmente uma nova estrutura de dados, caso seja detectada alguma oportunidade de melhorar a eficiência na utilização de recursos;

AP4 Desenvolver e especificar um novo algoritmo de inserção de dados, caso seja detectada alguma oportunidade de melhorar a eficiência na utilização de recursos ou devido a incompatibilidade do algoritmo original com uma eventual nova estrutura de dados;

QP2 **Existe algum modo de, durante a operação, inserir novos dados ou remover dados antigos da estrutura Nanocubes ou de alguma estrutura similar**?

AP5 Desenvolver e especificar um novo algoritmo que possibilite a remoção de dados da estrutura, uma vez que o Nanocubes não oferece esta possibilidade por utilizar apenas dados estáticos (*datasets*);

QP3 **É possível criar mecanismos para fazer a tecnologia ser adaptável, sendo capaz de receber a novos tipos de dados de entrada e produzir novos tipos de informação, bem como possibilitar a utilização de técnicas mais eficientes de produção de informação**?

AP6 Desenvolver e especificar um padrão para modularização que facilite o acréscimo de novos algoritmos para operar com os dados recebidos, possibilitando a utilização de novos tipos de dados, como dados derivados de imagens, sons, etc.;

AP7 Desenvolver e especificar um padrão para modularização que facilite o acréscimo de novos algoritmos para extração de informação à partir dos dados recebidos, incluindo novos tipos de informação ou métodos mais eficientes para gerenciamento da informação;

QP4 **É possível criar uma tecnologia flexível, baseada com uma estrutura de dados eficiente, que seja capaz de responder a solicitações de consulta de variados tipos**?

AP8 Desenvolver e especificar um padrão de comunicação envolvendo a especificação de alguma forma de linguagem para o recebimento de consultas e a devolução de respostas;

AP9 Desenvolver e especificar um processador de consultas agnóstico, capaz de lidar com os tipos de dados e informações operadas pela estrutura sem nenhum viés;

1.3 Objetivo

O objetivo principal deste trabalho é apresentar uma solução tecnológica capaz de atender simultaneamente a todos os requisitos dispostos na Seção 1.1. Como forma de atingir a este objetivo, foi desenvolvida uma nova tecnologia, denominada Tinycubes, que atende a todos requisitos dispostos na Seção 1.1 completa e simultaneamente. A tecnologia tem como núcleo uma nova estrutura de dados também denominada de Tinycubes. Essa nova estrutura, baseada na estrutura Nanocubes, considerada o estado-da-arte na sua área, foi concebida para ser tão ou mais eficiente na utilização de recursos do que a estrutura na qual foi baseada.

1.4 Organização deste documento

Este documento é organizado da seguinte forma: O Capítulo 2, Prolegômenos, contém seções que apresentam conceitos e definições relevantes para o entendimento do restante do trabalho. O Capítulo 3 discorre sobre trabalhos relacionados ao objetivo principal desta dissertação, analisando resultados de pesquisas referentes ao gerenciamento de dados para exploração visual e interativa em geral e, mais detalhadamente, os trabalhos relacionados à estrutura Nanocubes. O Capítulo 4 apresenta análises da estrutura Nanocubes (*AP1*) e do algoritmo de inserção (*AP2*) quanto a eficiência. O Capítulo 5 apresenta uma visão geral da tecnologia Tinycybes, introduzindo os conceitos essenciais para sua compreensão. No Capítulo 6, a especificação detalhada da estrutura de dados Tinycubes é apresentada(*AP3*), bem como são descritos os novos algoritmos de inserção (*AP4*) e de remoção de dados (*AP5*) da estrutura. O Capítulo 7 refere-se aos mecanismos de modularização oferecidos pela tecnologia que são utilizados pela estrutura de dados e pelos demais elementos da tecnologia, permitindo extensões ou especializações desses elementos (*AP6*, *AP7*). O Capítulo 8 é composto por seções que descrevem elementos da tecnologia Tinycubes que não fazem parte da estrutura de dados, como a especificação do Schema, a linguagem de consulta (*AP8*) e o processador de consultas agnóstico (*AP9*). No Capítulo 9 são apresentados resultados obtidos com a execução de experimentos sobre datasets sintéticos e datasets reais, bem como a apresentação de um caso de uso da tecnologia Tinycubes. O Capítulo 10 contém a conclusão do trabalho, apresentando considerações finais, contribuições realizadas, as limitações da tecnologia e possibilidades de desenvolvimento de trabalhos futuros.

Capítulo 2

Prolegômenos

As seções deste capítulo apresentam brevemente noções e conceitos que serão importantes para pleno entendimento dos demais capítulos deste trabalho. A primeira seção apresenta a tecnologia OLAP [9], datacubes e conceitos afins. A segunda seção discorre sobre a área de exploração visual e interativa de dados [18]. A terceira seção apresenta conceitos e definições baseados na Teoria dos Grafos. A quarta seção discorre sobre "tiles", um sistema de coordenadas que utiliza números inteiros. A quinta seção apresenta a estrutura de dados Quadtree [23], capaz de indexar pontos dispostos no espaço.

2.1 OLAP, Datacubes, Dimensões e Hierarquias

Bancos de dados tradicionais foram projetados com o objetivo principal de centralizar o armazenamento de dados que devem ser recuperados posteriormente sem que haja nenhuma degradação [28]. Esses bancos, que tipicamente são utilizados por diversos usuários e sistemas simultaneamente, são classificados como Bancos de Dados (ou sistema de informação) OLTP (*On-Line Transaction Processing*) [9]. Embora projetados visando o armazenamento e recuperação de dados operacionais, bancos de Dados OLTP também costumam oferecer facilidades para produção de informações analíticas decorrentes de agregação de dados, tais como contagem, totais e médias de dados armazenados.

Por outro lado, Bancos de Dados OLAP (*On-line Analytical Processing*) [9] foram especificamente projetados para facilitar a análise de dados através de consultas que produzam informações referentes a agregação desses dados, como totais e médias. Por conta desta característica, esses bancos são internamente organizados para reduzir o tempo necessário para realização de consultas analíticas, possivelmente armazenando as informações já agregadas, e não preservando registros de dados como ocorre num típico banco

OLTP. Bancos OLAP são ostensivamente utilizados em sistemas de suporte à decisão (DSS - Decision Suport Systems) [61, 62, 57] e aplicações de *Business Inteligence*, pois oferecem aos tomadores de decisão, como gerentes e administradores, facilidade e rapidez para extrair informações à partir do conjunto de dados operacionais a que tem acesso.

A organização interna de um Banco OLAP apresenta-se como um **datacube** (ou **OLAP cube** ou mesmo "data cube"), uma forma de armazenamento que acelera o processo de análise de dados e produção de informação [27, 29, 16]. Embora chamado de **cube** (cubo), o que remete a uma figura tridimensional, um **datacube** deve ser entendido como um hipercubo, podendo conter um número de lados arbitrariamente grande, ou mesmo pequeno, tendo um ou dois lados.

Um datacube é construído para que cada lado do cubo contenha valores calculados à partir de atributos relevantes para análise. Ao se definir uma tupla, na qual cada posição é preenchida com um valor obtido de cada lado, cria-se um sistema de coordenadas, que permite o rápido acesso às informações (normalmente contagens, totais ou outras medidas estatísticas) calculadas e armazenadas no cubo.

	2010	2011	2012	2013	2014
Português					
Matemática					
Ciências					
História					
Geografia					

(a)

	2010	2011	2012	2013	2014	*Todos*
Português						
Matemática						
Ciências						
História						
Geografia						
Todos						

(b)

Figura 2.1: Datacubes bidimensionais

Por exemplo, o diretor de uma escola poderia criar um datacube com apenas duas dimensões (como numa tabela), onde no eixo vertical ficariam as disciplinas e no eixo horizontal ficariam alguns anos referentes a resultados de cada disciplina. Em cada elemento desse datacube, identificado pela combinação (coordenada) (disciplina; ano), seriam armazenados o número de alunos e a média das notas. No final do ano atual, cada elemento com coordenada (disciplina; ano-atual) seria atualizado com as medidas estatísticas calculadas à partir das notas de cada aluno. Assim, quando o diretor quiser obter informações sobre uma disciplina para um determinado ano, apenas seria necessário consultar o elemento na coordenada (disciplina; ano) correta e coletar as informações armazenadas nele. A Figura 2.1 (a) mostra um possível datacube correspondente a este exemplo.

Seguindo ainda nesse exemplo, observe que se o diretor desejasse estatísticas referentes a todo um ano, independentemente da disciplina, seria necessário percorrer a coluna

correspondente ao ano, combinando o resultado de todos os elementos para obter a informação desejada. Obviamente isso geraria um esforço adicional durante a consulta quando comparada a consulta específica sobre (disciplina; ano). Como forma de tornar a recuperação da informação mais ágil, seria possível acrescentar uma linha a mais no datacube, que armazenaria, para cada coluna, os valores resultantes da pré-agregação de todas as linhas à medida em que novos dados são introduzidos. Os elementos desta linha teriam a coordenada, (*Todos*; ano), indicando que a informação refere-se a todas as disciplinas daquele ano. Assim, quando o diretor quisesse informações referentes a todas as disciplinas para um ano, precisaria apenas consultar o elemento (*Todos*; ano). O mesmo processo poderia ser aplicado à coluna correspondente aos anos. É possível criar uma nova coluna com coordenadas (disciplina; *Todos*) que armazenaria a pré-agregação dos valores de todas os elementos de uma disciplina, a despeito do ano. Note que, seguindo este padrão, seria criado um elemento com coordenada (*Todos*; *Todos*), que armazenaria a informação pré-agregada de todas as disciplinas e todos os anos. A Figura 2.1(b) ilustra o resultado após o acréscimo da linha e da coluna *Todos*.

Em um datacube, é comum existirem informações pré-agregadas referentes a todos os elementos de uma linha. Isso acontece justamente para acelerar a recuperação das informações, minimizando o número de operações a serem realizadas durante à consulta. No entanto, essa adição de novas "linhas" tem o custo de aumentar significativamente a quantidade de elementos armazenados. Se considerarmos que o número de elementos num datacube pode ser representado por um produto cartesiano, no qual cada fator é dado pelo número de elementos, então, ao se adicionar uma coluna a mais em cada "lado" do datacube, o efeito é significativo. Por exemplo, suponha que no exemplo anterior existiam 10 disciplinas (linhas) e 5 anos, o que geraria 50 elementos no total. Ao se acrescentar uma coluna a cada lado da tabela, o número de elementos sobe para 66, o que corresponde a um aumento de mais de 20%. Se imaginarmos que o colégio faz parte de uma rede com 4 unidades, deveria ser criado um novo lado com 4 posições, elevando o número de elementos do datacube para 200 ($10 \times 5 \times 4$). Ao se utilizar o valor *Todos*, o número de elementos passaria a ser 330 ($(10 + 1) \times (5 + 1) \times (4 + 1)$), o que corresponde a um acréscimo de 65%.

Um outro exemplo envolvendo uma empresa revendedora de produtos que atua em diversas cidades, pode-se construir um datacube com três lados, onde um lado do cubo estaria associado a identificação dos produtos do catálogo, outro lado às cidades onde a empresa atua e o último lado a um mês/ano, gerando a coordenada (produto; cidade; mês/ano). Nesse cenário, quando uma venda de produto de catálogo é realizada, o ele-

mento do datacube, cuja coordenada é dada pelo produto, cidade onde a venda ocorreu e mês/ano da venda, é devidamente atualizado (incrementando-se o total de vendas, por exemplo). Posteriormente, quando se deseja saber o total de vendas para uma combinação de produto, cidade e mês/ano, basta o banco OLAP consultar diretamente o elemento definido por essas coordenadas para obter a informação desejada, não sendo necessário realizar pesquisas e cálculos sobre conjuntos de dados. Essa agilidade para acessar informações pré-calculadas é o que possibilita o baixo tempo de resposta (baixa latência) para consultas analíticas.

Em um datacube, uma **dimensão** corresponde à forma como os usuários pensam sobre um valor utilizável pelas consultas. Uma dimensão pode ser simples, o que corresponderia diretamente a um lado de um cubo. No exemplo anterior, tanto a identificação de um produto quanto a cidade onde a venda ocorreu seriam consideradas dimensões simples. Por outro lado, o mês/ano de venda é uma dimensão composta, dado que é formada por dois valores. Note que, em um datacube, uma dimensão composta por mais de um valor sempre é **hierárquica**, onde um valor conceitualmente controla o valor de outro subordinado. No caso, o valor do ano, por exemplo, 2019, conceitualmente controla o valor do mês (janeiro/2019). Observe que, dependendo da forma como se deseja pensar sobre os dados, uma dimensão pode ser simples ou não. Por exemplo, se a empresa desejasse atuar em vários estados, então as cidades, que originalmente eram dimensões simples, poderiam passar a ser representadas como estado/cidade, e portanto passariam a estar subordinadas a um estado.

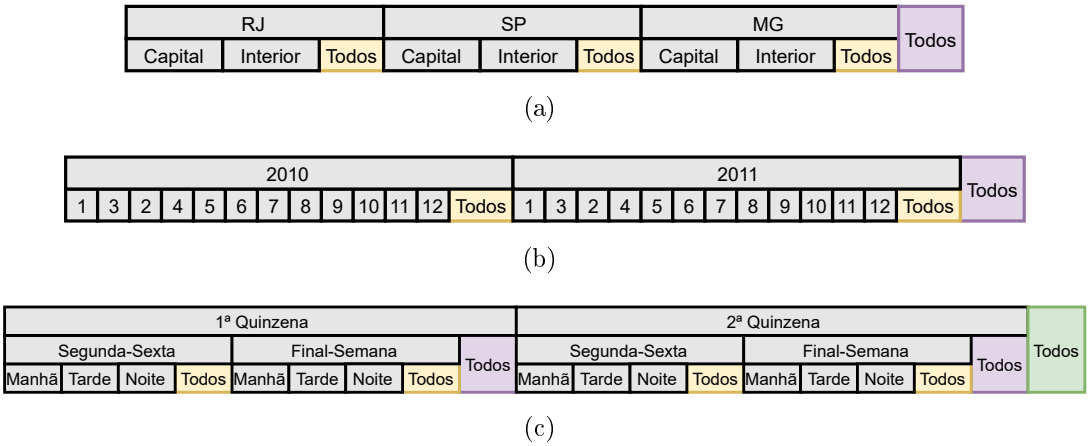


Figura 2.2: Três exemplos de dimensões hierárquicas

A Figura 2.2 contém três exemplos de dimensões hierárquicas. Na parte (a), vê-se uma dimensão hierárquica espacial que na parte mais geral exibe estados e o nível mais detalhado divide-se em "Capital" e "Interior". A parte (b) contém uma hierarquia temporal

cuja parte geral é o ano e a detalhada, o mês. Já a parte (c) também exibe uma dimensão hieráquica temporal, mas com dois detalhes diferentes. O primeiro detalhe consiste no fato de que existem três níveis de detalhamento, e portanto tem de existir três níveis de colunas agregadoras *Todos*, como aparece na parte mais a direita da figura. O segundo detalhe é que qualquer registro de tempo pode ser armazenado nesta dimensão hierárquica, visto que ela não contém vínculos absolutos como anos. Isso ilustra a liberdade que o projetista de um datacube pode ter ao converter dados recebidos em dimensões úteis para análise que deseja realizar.

As **dimensões hierárquicas** permitem a realização de consultas com diferentes níveis de detalhamento. Ainda usando o exemplo anterior, poderia-se realizar consultas especificando-se apenas o ano, para se ter informações mais condensadas, ou especificando o mês/ano, para se ter informações mais detalhadas. É importante ressaltar que um mesmo tipo de valor pode ser utilizado em diferentes hierarquias. Considere que os produtos do exemplo possam ser agrupados, tanto de acordo com o país de origem quanto de acordo com o segmento que são utilizados, como produtos de beleza ou eletrodomésticos. Utilizando uma organização como essa, seria possível criar duas dimensões hierarquias diferentes onde um mesmo produto estaria subordinado.

2.2 Exploração visual e interativa de dados - EVID

A exploração visual e interativa de dados é conhecida na literatura por diversos termos em inglês: "*Data Analisys*" [74], "*Visual Analytics*" [18], "*Visual Exploration*" [73], "*Interactive Visual Exploration*" [70], "*Real-Time Visual Exploration*" [56], etc. Dada a variedade de opções disponíveis para nomear esta área de pesquisa, preferiu-se utilizar neste trabalho um nome que identificasse claramente a ideia principal: a exploração dos dados disponíveis utilizando métodos visuais e interativos. No entanto, não se pode perder de vista que dados não se constituem automaticamente em informação, portanto, o processo de exploração envolve a extração de informação de subconjuntos dos dados disponíveis.

Nesta seção são apresentados conceitos, requisitos e desafios tecnológicos inerentes à exploração visual e interativa de dados (EVID), com ênfase na utilização de grandes volumes de dados, possivelmente contendo informações espaciais e temporais. Esta seção se divide em duas partes, os requisitos necessários para uma experiência adequada de EVID e os desafios decorrentes de tentar atender a esses requisitos.

2.2.1 Requisitos para EVID

Devido às limitações cognitivas dos seres humanos [49] que tornam quase impossível perceber informações relevantes através da inspeção direta de dados coletados, ao longo do tempo foram desenvolvidas técnicas que possibilitaram a extração de informações relevantes à partir desse dados. Algumas dessas técnicas baseiam-se na transformação dos dados coletados em medidas estatísticas, como totais, médias, desvios padrão e distribuições de frequência [48, 2]. Outras técnicas baseiam-se na representação dos dados de forma gráfica e visual, com o uso recorrente de símbolos [5]. Para aumentar as capacidades de apresentação de informação, essas duas famílias de técnicas costumam ser utilizadas em conjunto, onde medidas estatísticas são apresentadas de forma gráfica [24].

Atualmente, técnicas de visualização de dados oferecem a possibilidade de uma mesma imagem exibir diferentes informações simultaneamente apenas através da utilização de recursos gráficos. Trabalhos focados na área de visualização de informação [21, 47], identificaram diversas dimensões para transmissão de informação através de imagens, tais como, forma, cor, tamanho, orientação, textura, etc. [40].

No entanto, o processo de selecionar quais informações são relevantes para tomada de decisão nem sempre é simples [70, 65]. O processamento computacional permite a extração de diferentes categorias de informação de dados coletados, que ainda não sofreram nenhuma transformação. Cabe ao tomador de decisão, descobrir, por vezes através de um processo exploratório de tentativa e erro, quais informações podem ser consideradas relevantes e até mesmo quais dados devem ser utilizados para sua produção [3]. Este processo exploratório, onde se avalia a relevância dos dados disponíveis e das informações geradas por eles, tem se tornado mais complexo, justamente devido à riqueza e ao volume dos dados e consequentemente das informações extraídas deles por processos computacionais [52]. O artigo seminal "The Eyes Have It: A Task by Data Type Taxonomy for Information Visualizations" discute as operações tipicamente envolvidas na exploração de dados [63].

Hoje em dia, com a evolução dos computadores, tornou-se possível produzir gráficos que representam conjuntos de dados ou de informações em intervalos de tempo cada vez mais reduzidos [53]. Essa redução foi tão significativa que, em diversas situações, já é possível gerar esses gráficos de forma interativa, ou seja, de forma que a latência, tempo decorrido entre a requisição do gráfico e o término da sua exibição, tem a mesma ordem de grandeza que a latência em um diálogo entre dois adultos. Por outro lado, já foi constatado [42] que, se o tempo de latência for superior a 500 ms, o processo exploratório

torna-se menos efetivo, com aumento da inatividade dos usuários.

Ao mesmo tempo, a disseminação e miniaturização de computadores, que viabilizaram a criação de dispositivos IoT, oferecem mais fontes geradoras de dados coletáveis. Em diversos cenários, como no caso de monitoramento de serviços de atendimento à população, gestores devem ter acesso a informações que provavelmente serão produzidas pela análise de séries históricas de dados de monitoramento, oriundos de diferentes regiões geográficas. Esses dados são denominados de "geo-temporais" ou "espaço-temporais".

2.2.2 Alguns desafios

O desenvolvimento de tecnologias que possibilitem a exploração visual e interativa de dados (EVID) para grandes volumes de dados complexos gerados continuamente apresenta desafios em diversas áreas da informática. Do ponto de vista visual, a escolha da forma mais adequada para apresentação das informação desejadas se constitui numa área de pesquisa bastante ativa [74, 41, 22, 21, 18]. A produção de informações relevantes à partir de dados coletados também oferece oportunidades para o desenvolvimento de novos algoritmos e abordagens [22]. Outra categoria de pesquisa, e o foco deste trabalho, refere-se ao desenvolvimento de tecnologias para o gerenciamento de dados na qual consultas que demandem informações extraídas de grandes volumes de dados armazenados sejam respondidas em tempo suficientemente curto para permitir a exploração interativa, com baixa latência entre a solicitação de consultas e a apresentação das respostas.

A criação de tecnologias para o gerenciamento de dados, no contexto da EVID, apresenta dois desafios importantes. O primeiro desafio decorre do grande volume dos dados armazenados [13]. É relativamente trivial armazenar, para fins de extração rápida de informação, pequenos volumes de dados complexos utilizando formas de armazenamento convencionais como matrizes, arquivos de dados e até mesmo bancos de dados relacionais. No entanto, à medida que o volume de dados cresce, o tempo necessário para extração de informações tende a crescer a ponto de tornar a interação inviável.

O segundo desafio origina-se na universalização e da miniaturização dos computadores [4, 25]. Tradicionalmente, sistemas para EVID obtém informações à partir de dados estáticos potencialmente antigos, que podem não refletir mais a situação atual. Com a melhoria na infraestrutura de telecomunicações e a oferta continua de dados coletados por equipamentos computadorizados e dispositivos IoT, os sistemas de EVID têm à sua disposição dados recém coletados, o que permite a produção de informações que podem refletir melhor o tempo presente [66]. No entanto, a necessidade de armazenar continua-

mente esses novos dados pode levar ao esgotamento do espaço de armazenamento. Desta forma, surge o desafio de criar mecanismos que evitem a paralisação da análise por falta de espaço de armazenamento.

2.3 Record como conjunto

Records ("registros", em português) são elementos típicos da Ciência da Computação que, às vezes, podem ser utilizados em definições matemáticas. Tipicamente um Record é uma sequência finita e ordenada de campos (variáveis) nomeados, sendo que cada um deles armazena "valores" de um determinado "Tipo", considerando que um "valor" pode ser entendido como um elemento de um conjunto (o "Tipo"), este último possuindo propriedades bem definidas.

Neste trabalho, cada estrutura de dados correspondente a um Record poderá ser utilizada como um conjunto no sentido matemático. Os elementos que compõem esse conjunto são todas as possíveis instâncias da estrutura de dados que define o Record. Por exemplo, considere um Record definido como:

$$Pessoa = \{nome : Texto; \quad idade : Inteiro\}$$

no qual existem dois campos, "nome" e "idade", que podem armazenar, respectivamente, valores textuais e inteiros. Portanto, como Records são considerados conjuntos, o Record Pessoa pode ser utilizado para definir funções matemáticas. Como por exemplo:

$$idoso : Pessoa \mapsto \{ \text{ "True", "False" } \}$$

cuja implementação poderia retornar "True" apenas se o valor campo "idade" da instância de Record (elemento do conjunto) aplicado à função for maior ou igual a 64 anos. Por exemplo, a função *idoso* retornaria "True" e "False" quando aplicada, respectivamente, às instâncias $\{nome : \text{"Alfredo"}; \quad idade : 72\}$ e $\{nome : \text{"Pedro"}; \quad idade : 32\}$.

2.4 Teoria dos Grafos

A teoria dos grafos é um ramo da matemática que oferece meios para representar formalmente relações entre objetos matemáticos ou objetos reais modelados matematicamente. A partir dessa representação formal, é possível, não apenas identificar relações existentes, como também descobrir relações ocultas pela complexidade inerente às situações repre-

sentadas.

Utiliza-se a teoria dos grafos para modelar (com o significado de representar algo respeitando regras precisas), não só uma grande variedade de elementos do mundo real, como redes de computadores, moléculas, malhas ferroviárias etc., mas também o próprio conhecimento, como modelos de dados e modelos de informação para bancos de dados.

Nesta seção será realizada uma breve apresentação dos conceitos e definições da Teoria dos Grafos relevantes para a especificação e entendimento da estrutura de dados Tinycubes. Para uma descrição mais ampla, detalhada e ainda mais formal da teoria dos grafos consulte textos específicos como [44, 17] [1].

Definição 2.4.1. grafo: Um grafo é uma estrutura matemática formada por um conjunto finito não vazio V de vértices e um conjunto finito E (*Edges*, em inglês) de arestas, descrito formalmente como:

$$G =< V, E > \tag{2.1}$$

Alternativamente, dado um grafo G, pode-se representar o conjunto V e o conjunto E como:

$$V = V(G) \tag{2.2}$$

$$E = E(G) \tag{2.3}$$

Definição 2.4.2. ordem e tamanho de um grafo: A ordem de um grafo G é dada pelo número de vértices do grafo, $|V(G)|$. O tamanho de um grafo G é dado pelo número de arestas do grafo, $|E(G)|$.

Definição 2.4.3. vértice: Um vértice v ($v \in V$) é qualquer objeto de interesse que pode ser modelado como elemento de um conjunto. Um vértice também pode ser chamado de "nó" (*node*, em inglês). Graficamente, num plano, os vértices são representados como pontos ou outras figuras geométricas fechadas, tais como círculos, quadrados, triângulos, etc.

Definição 2.4.4. aresta ou aresta não direcional: Uma aresta a ($a \in E$) é um **par não ordenado** de vértices, sendo matematicamente descrita como $a = (v_1, v_2)$ onde $v_1, v_2 \in V$. Por ser um par não ordenado, na prática, se $a = (v_1, v_2)$ então $a = (v_2, v_1)$. Graficamente, as arestas são representados como segmentos de linhas que ligam dois vértices.

[1] Existem formas alternativas para definição de alguns conceitos da Teoria dos Grafos. Neste trabalho foi escolhida a forma que se revelou mais adequada.

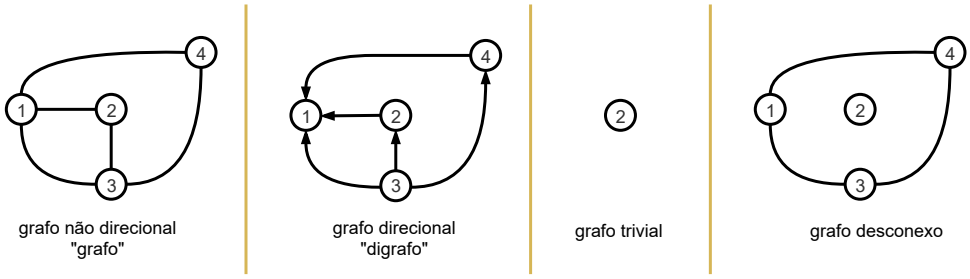


Figura 2.3: Representação gráfica para grafos

Definição 2.4.5. aresta direcional: Uma aresta direcional a ($a \in E$) é um **par ordenado** de vértices, sendo matematicamente descrita como $a = (v_o, v_d)$ onde $v_o, v_d \in V$. Os vértices v_o e v_d são denominados, respectivamente, **vértice origem** e **vértice destino**. Arestas direcionais também podem ser chamadas de setas ou de arcos. As arestas direcionais são representadas graficamente como segmentos de linhas que ligam dois vértices com alguma "marca" na extremidade próxima ao vértice destino, usualmente, uma seta.

Quando uma aresta a tem um vértice v_o como vértice origem, pode se dizer que "**a tem origem em v_o**", ou que "a se origina em v_o" ou ainda que "v_o é origem de a". Por outro lado, quando v_d é o vértice destino de uma aresta a, costuma-se afirmar que "**a aponta para v_d**" ou alternativamente que "v_d é apontado por a", ou que "v_d é atingido por a" ou ainda que "v_d é destino de a".

Definição 2.4.6. grafo direcional - digrafo: Um digrafo é um grafo onde todas as arestas são arestas directionais.

Definição 2.4.7. grafo não direcional: Um grafo não direcional é um grafo onde todas as arestas não são arestas directionais, ou seja, é a definição básica de um grafo.

Definição 2.4.8. grafo trivial: Um grafo trivial é um grafo com um único vértice ($V = \{v_0\}$) e nenhuma aresta ($E = \emptyset$).

Definição 2.4.9. caminho: Informalmente, um caminho é uma forma de, à partir de um vértice x, atingir um vértice y ausando as arestas do conjunto E. Formalmente, um caminho é uma sequência de n ($n \in \mathbb{N}^+$) arestas e $(n+1)$ vértices no formato $(v_0, a_1, v_1, a_1, v_2, ..., a_n, v_n)$, onde $v_0 \in V$ e $\forall i \in [1, n]$, $a_i \in E$, $a_i = (v_{i-1}, v_i)$ ou $a_i = (v_i, v_{i-1})$ e $v_i \in V$.

Definição 2.4.10. caminho dirigido: A definição é similar ao caminho definido em 2.4.9, porém usando arestas dirigidas. Formalmente, um caminho é uma sequência de n ($n \in \mathbb{N}^+$) arestas e $(n+1)$ vértices no formato $(v_0, a_1, v_1, a_2, v_2, ..., a_n, v_n)$, onde $v_0 \in V$ e $\forall i \in [1, n]$, $a_i \in E$, $\boldsymbol{a_i = (v_{i-1}, v_i)}$ e $v_i \in V$.

Note que a definição de caminho, independentemente de ser dirigido ou não, permite que um mesmo vértice possa aparecer mais de uma vez na sequência que compõe o caminho, pois o índice utilizado na definição formal refere-se apenas a posição na sequência e não a identidade do vértice ou da aresta em si mesmo. A definição à seguir criará uma restrição para a possibilidade de reuso do vértice em um caminho.

Definição 2.4.11. caminho simples: Um caminho simples é um caminho onde nenhum vértice ocorre mais uma vez na sequência.

Definição 2.4.12. vértice pai: Um vértice v é considerado pai de uma aresta direcional a ($a \in E$) e do vértice destino da aresta a, se é o vértice origem de a.

Definição 2.4.13. vértice filho: Um vértice y é considerado vértice filho de uma aresta direcional a ($a \in E$) ou de um vértice x se é destino da aresta a ou existe alguma aresta direcional formada pelo par (x,y), respectivamente.

Definição 2.4.14. vértice descendente: Num digrafo, um vértice y é descendente de um vértice x se existe um caminho simples iniciado no vértice x e terminado no vértice y.

Definição 2.4.15. aresta descendente: Num digrafo, uma aresta a é descendente de um vértice x se existe um caminho simples iniciado no vértice x no qual a aresta a faz parte.

Definição 2.4.16. aresta irmã: Uma aresta direcional a ($a \in E$) é considerada irmã de outra aresta direcional b ($b \in E$) se ambas tem o mesmo vértice v como vértice origem.

Definição 2.4.17. comprimento de um caminho: O comprimento de um caminho é número de arestas que compõem um caminho.

Definição 2.4.18. grafo conexo (em um grafo não direcional): Um grafo é conexo quando, à partir de um vértice qualquer, todos os demais vértices podem ser atingidos usando-se as arestas em E.

Definição 2.4.19. digrafo conexo ou digrafo fracamente conexo (em um digrafo): Um digrafo é classificado como conexo (ou fracamente conexo) quando, à partir de um vértice qualquer, todas os demais vértices podem ser atingidos usando-se **versões não direcionais** das arestas em E. Dito de outra forma, primeiro transforma-se o digrafo em um grafo não direcional e depois usa-se a definição de grafo conexo para grafos não direcionais.

Definição 2.4.20. subgrafo: Informalmente, H é um subgrafo de um grafo G se os vértices e as arestas de H estão em subconjuntos dos conjuntos de vértices e de arestas de G. H é dito subgrafo de G se:

- $V(H) \subseteq V(G)$
- $E(H) \subseteq E(G)$

Definição 2.4.21. Árvore. Uma árvore é grafo não dirigido conexo G onde sempre existe um único caminho interligando dois vértices distintos.

Definição 2.4.22. Arborescência. Uma arborescência é um grafo dirigido G contendo um vértice v_0, chamado **raiz**, tal que, dado qualquer outro vértice v em G, sempre existe um único caminho partindo de v_0 e chegando a v.

2.5 Grafos e Árvores como estruturas de dados

Esta seção apresenta alguns os conceitos referentes à grafos e árvores de busca, que são necessários para o entendimento da estrutura de dados Tinycubes.

Grafos em estruturas de dados

Na Teoria dos Grafos, um grafo é composto por uma organização específica de vértices interligados por arestas. Ao se acrescentar um novo vértice ou aresta a um grafo A, cria-se um novo grafo B. Por outro lado, quando uma estrutura de dados implementa um grafo A, considera-se que a inserção e remoção de algum elemento neste grafo apenas o modifica, não criando nenhum novo grafo. Essa preservação da identidade do grafo após modificações é importante na prática porque garante consistência conceitual à variáveis nomeadas que referenciam grafos modificados ao longo da execução de um programa.

A preservação da identidade de um grafo após modificações também é relevante para especificação de estruturas de dados modeladas por grafos, como no caso do Tinycubes, porque permite descrever da estrutura como um grafo que se modifica ao longo do tempo de execução. Desta forma é possível especificar a organização da estrutura através de regras e propriedades referentes ao Grafo que a modela.

Definição 2.5.1. Grafo (estrutura de dados): No contexto de estruturas de dados, um Grafo, pode ser alterado através do acréscimo e remoção de vértices ou arestas, desde que possua algum elemento que permita identificá-lo.

Árvores de busca em estruturas de dados

O conceito de árvore na Teoria dos Grafos (definição 2.4.21) é baseado em arestas não dirigidas, ou seja, arestas que não distinguem seus vértices como origem e destino. Por

outro lado, arborescências (definição 2.4.22) são baseadas em arestas dirigidas e, portanto, dada uma aresta pertencente a um caminho, sabe-se qual vértice a precede e qual a sucede. Como vermos a seguir, esta característica é muito útil para criação de estruturas de dados eficientes para localização seletiva de informações, porém antes é necessário realizar um ajuste conceitual.

Definição 2.5.2. Árvore (estruturas de dados): Uma árvore, no contexto de estruturas de dados, equivale à arborescência definida pela teoria dos grafos. Da mesma forma que em uma arborescência matemática, numa árvore existe um nó (vértice) **raiz**, à partir da qual pode-se alcançar qualquer outro nó seguindo através das arestas. Além disso, considera-se que inserções e remoção de vértices e arestas em uma árvore com um mesmo vértice raiz, não criam uma nova árvore.

De fato, as estruturas de dados implementadas com base em ponteiros geralmente são baseadas em grafos dirigidos onde um ponteiro é uma forma de implementar uma aresta direcional.

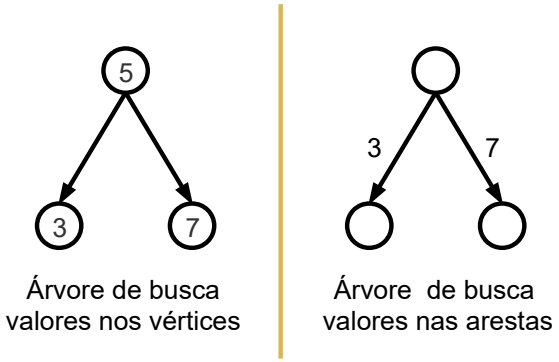


Figura 2.4: Representação gráfica para árvores

Definição 2.5.3. árvore de busca: Uma árvore de busca é uma árvore capaz de armazenar valores de forma a reduzir o esforço computacional para localizá-los posteriormente. Normalmente os valores são armazenados nos vértices de uma árvore, embora seja possível definir árvores onde os valores são armazenados nas arestas, como ilustra a Figura **??** . Independentemente de onde são armazenados os valores utilizados numa árvore de busca, sempre é necessário organizar as arestas para criação de algoritmos capazes de acessar os valores de forma rápida.

Definição 2.5.4. árvores binária, **n-ária**, **não binária**: Uma árvore de busca é denominada árvore binária quando um nó (vértice) pode ser origem de até duas arestas, identificadas como aresta esquerda e aresta direita; árvore n-ária, quando um nó pode ser

origem de até n arestas e; árvore não binária, quando um nó pode ser origem de mais do que duas arestas, porém sem limite superior definido. Nas árvores os nós podem ter mais de duas arestas, normalmente cada aresta é identificada por um número inteiro não negativo sequencial.

Uma árvore de busca define uma partição

As árvores de busca reduzem o esforço para localização de valores porque são construídas para que vértice leve a um dos subconjuntos de uma partição do espaço de busca. Tipicamente, uma árvore armazena valores de um domínio ordenável[2] D onde cada vértice v_i é responsável por um intervalo I_i dos valores de D, sendo que os vértices descendentes de v_i são responsáveis por intervalos que não se superpõem e que, ao serem unidos (no sentido de teoria dos conjuntos), correspondem exatamente ao intervalo I_i. Dito de outra forma, cada vértice de uma árvore de busca corresponde a uma segmentação de D.

2.6 Tiles - visualização de mapas

O sistema de coordenadas geográficas baseado em latitude e longitude apresenta algumas dificuldades quando utilizado para exibir mapas na tela de um computador devido, sobretudo, a estar projetando uma superfície quase esférica num plano [11]. O cálculo correto da projeção numa área contendo uma quantidade de *pixels* variável conforme o tamanho da janela de visualização pode se tornar complexo, demorado e demandar recursos computacionais significativos.

Para agilizar a exibição de mapas em telas de computadores, utiliza-se uma técnica onde a um mapa, resultado de alguma forma de projeção da superfície da Terra em um plano, é dividido, tanto horizontalmente quanto verticalmente, em pequenos pedaços de igual tamanho denominados "**Tiles**" [59] [46] (azulejos em português). De fato, cada tile é um bitmap com tamanho fixo, normalmente contendo 256 x 256 *pixels*, correspondentes à representação gráfica da parte do mapa que lhe foi atribuída. Cada um desses tiles é numerado conforme sua posição horizontal e vertical em relação ao mapa, possibilitando a utilização desta numeração para gerar um sistema de coordenadas de valores discretos, o que simplifica o acesso a pedaços de imagens pré-computadas do mapa. É importante perceber que a quantidade de divisões do mapa define o nível detalhe que os tiles irão oferecer, uma vez que cada tile tem um tamanho limitado de *pixels*.

[2] Conjunto no qual existe uma relação de ordem

Para visualização do mapa da Terra, a quantidade de tiles é dada por uma potência de 2, calculada como 2^z, onde z é chamado de "zoom". Assim, quando o zoom é zero (0), um único tile cobre todo o mapa. Com zoom = 1, têm-se 4 tiles (2 na vertical e 2 na horizontal), com zoom=2, 16 tiles e assim por diante. Com o nível de zoom adequado, pode-se utilizar o sistema de coordenadas baseado em tiles para localizar objetos com algo grau de precisão.

A título de ilustração, as fórmulas a seguir descrevem como realizar a conversão do sistema de coordenadas baseado em latitudes e longitudes para o sistema baseado em tiles.

$$tileHorizontal = \frac{2^{zoom}(longitude + 180)}{360} \tag{2.4}$$

$$tileVertical = \frac{2^{zoom-1}(longitude + 90)}{180} \tag{2.5}$$

2.7 Quadtrees

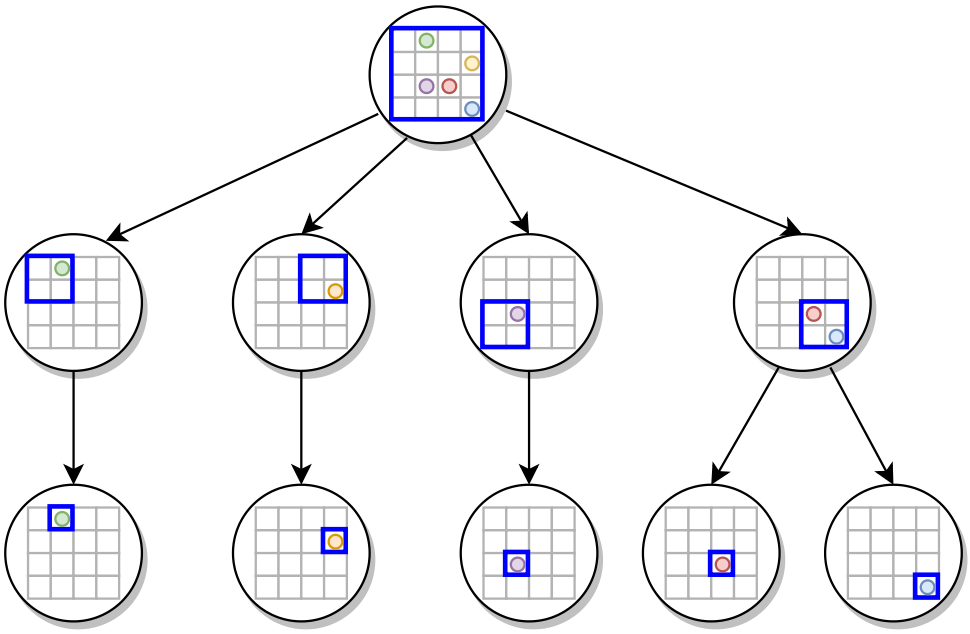


Figura 2.5: Quadtree indexando 5 subáreas com precisão de 1/16 da área total

Quadtree [23] é uma estrutura de dados tipo árvore de busca criada para acelerar a localização de elementos que podem ser identificados por chaves compostas, como locais definidos por coordenadas geográficas bidimensionais.

Uma Quadtree bidimensional possibilita a localização de elementos com uma precisão controlável. Quando utilizada com um plano bidimensional, cada vértice v da árvore representa uma região r do plano e cada um dos eventuais quatro vértices atingíveis à

partir de v representa um quadrante da região r. Assim, à medida que se desce pela árvore, afastando-se da raiz, as regiões do plano representadas por um vértice são cada vez menores, correspondendo a um quadrante contendo 1/4 da área da região anterior. Desta forma, a altura da Quadtree determina a precisão para localização de elementos ao determinar o tamanho da menor área representável. Quando o plano utilizado é um mapa bidimensional da superfície da Terra, uma Quadtree que possui 25 arestas até as folhas, é capaz de representar áreas com $\frac{40.075Km}{2^{25}} = 1{,}2$ m de lado, aproximadamente.

Capítulo 3

Trabalhos Relacionados

Conforme discutido na Seção 2.2, a exploração visual e interativa de dados (EVID) é um amplo campo de estudo, permitindo o desenvolvimento de pesquisas nas área de: visualização de informação, o que inclui pesquisas sobre representação da informação, métodos de produção de imagens (*rendering*), etc.; desenvolvimento de métodos para extração de informação à partir de dados puros; criação de técnicas para o gerenciamento de dados que possibilitem armazenamento de grandes volumes de dados espaço-temporais complexos de modo que seja possível, subsequentemente, extrair rapidamente informações à partir de deles.

O estudo desenvolvido neste trabalho envolve a criação de uma tecnologia para gerenciar o utilização de grandes volumes de dados espaço-temporais complexos gerados continuamente, permitindo a produção de informações precisas à partir deles. Essa tecnologia é alicerçada em uma nova estrutura de dados, denominada Tinycubes, baseada no conceito de datacubes (Seção 2.1, [27, 29]) tendo sido inspirada na estrutura Nanocubes [38].

Este capítulo apresenta trabalhos com objetivos que atendem boa parte dos requisitos apresentados na Seção 1.1 e que definem o escopo do estudo desenvolvido, iniciando com uma rápida explanação sobre a área de gerenciamento de dados para EVID e posteriormente descrevendo com mais detalhes trabalhos baseados em datacubes e Nanocubes.

3.1 Gerenciamento de dados para EVID

Nesta seção serão apresentadas algumas estratégias para superar alguns desafios relativos à tecnologias para o gerenciamento de dados para EVID e que, portanto, seja capaz de

produzir respostas rápidas para solicitações de informação baseadas em grandes volumes de dados complexos.

Redução do volume de dados

Esta estratégia, usualmente identificada como **Approximate Query Processing** (AQP), consiste em reduzir o tempo de resposta das consultas através da redução artificial da quantidade de dados processados para produção de informação, onde parte do conjunto de dados disponíveis é ignorada. A consequência negativa óbvia do uso desta estratégia é a potencial perda de precisão na informação produzida. Ao deixar de ter acesso a parte dos dados disponíveis, os algoritmos para extração de informação podem produzir informações significativamente distintas daquelas computadas utilizando todos os dados.

Diferentes abordagens foram desenvolvidas com o objetivo de minimizar o impacto de se ignorar parte dos dados disponíveis através de métodos que esperam obter uma aproximação adequada das informações demandadas. Dentre essas abordagens, destacam-se a **amostragem estatística** [12, 54], utilizada no sistema BlinkDB [1], o uso de **wavelets** [8] e mais recentemente, a utilização de **redes neurais** como na estrutura de dados NeuralCubes [72], posteriormente rebatizada como NNCubes, onde uma rede neural é treinada para sintetizar respostas a consultas. Todas essas abordagens, apesar de eventualmente produzirem aproximações adequadas das informações desejadas, não serão discutidas em mais detalhes por estarem fora do escopo deste trabalho.

Utilização de recursos distribuídos

Esta estratégia baseia-se em aumentar as capacidades de processamento de grandes volumes de dados distribuindo o trabalho por diversos computadores em rede como em [19] ou utilizando um *middleware* que implementa um cache de dados distribuído, STASH [52]. Se por um lado, o volume de dados armazenados e a quantidade de consultas atendidas podem aumentar, o impacto na latência pode ser significativo, sobretudo quando as informações demandadas necessitam de dados que estão dispostos em mais de um computador.

Desenvolvimento de novas estruturas de dados

O desenvolvimento de estruturas de dados capazes de oferecer baixa latência na produção de respostas analíticas, mantendo um consumo moderado de recursos computacionais mesmo utilizadas com grandes volumes de dados, tem sido objeto de pesquisa intensa.

Estruturas de dados capazes de lidar com dados complexos (espaciais, temporais ou compostos) vem sendo desenvolvidas ao longo do tempo, tais como TeslecopicView-Tree(TV-Tree) [37] que permite a indexação de dados de diferentes tipos e Vistrees [20] projetada para rápida computação dinâmica de histogramas.

Aproveitando o fato de que a maior parte das consultas em EVID requisitam informações de cunho analítico, foram realizadas várias abordagens baseadas em bancos OLAP (Seção 2.1) como as apresentadas em [64, 58, 67, 45, 78], ou apenas utilizando o conceito de datacubes como em [36, 55, 14]. No entanto, abordagens baseadas em datacubes tendem a demandar grande quantidade de memória. Usualmente um datacube (Seção 2.1) armazena, para cada uma das combinação de valores dimensionais, um ou mais valores analíticos pré-computados que são utilizados para rápida obtenção de informações. O problema é que, à medida que a quantidade de valores dimensionais distintos aumenta, o número de combinações cresce rapidamente, inviabilizando a utilização de datacubes tradicionais com grandes volumes de dados.

Em resposta ao elevado consumo de memória decorrente do uso de datacubes, a estratégia **Iceberg cubes** [6] foi desenvolvida. Nesta abordagem, que vem inspirando diversos trabalhos [76, 60, 75, 36, 30], a pré-computação e armazenamento de valores agregados ocorre apenas se a quantidade de dados envolvida superar um determinado limite, reduzindo a quantidade de memória utilizada.

A estrutura de dados Nanocubes [38], também baseada em datacubes, tem se destacado ao apresentar bons resultados, equilibrando um baixo tempo de latência e um moderado consumo de memória. Esse equilíbrio possibilita a rápida extração de informações à partir de milhões de dados espaço-temporais, utilizando apenas os recursos computacionais encontrados em computadores de uso comum como, por exemplo notebooks. A estrutura criou estratégias e algoritmos para possibilitar o compartilhamento dos valores calculados, permitindo a redução da quantidade de memória necessária para armazenar valores pré-calculados (Seção 2.1) e consequentemente viabilizando a utilização de datacubes em EVID.

3.2 Nanocubes

Utilizando o conceito de datacubes, a estrutura Nanocubes definiu mecanismos para compartilhamento sistemático de valores agregados pré-computados, como utilizados em cubos OLAP (ver Seção 2.1). Devido a esse compartilhamento, a quantidade necessária

de memória RAM para o armazenamento desses valores pré-computados pode ser reduzida nas situações previstas pelo algoritmo de inserção de dados utilizado pela estrutura Nanocubes.

Por outro lado, a estrutura Nanocubes não oferece nenhum algoritmo para remoção dos valores armazenados[*Req10*, definido previamente na Seção 1.1], ou seja, ao se utilizar Nanocubes, a única forma de remover algum valor armazenado implica na remoção de todos os valores armazenados com a subsequente reinserção de todos os valores que não deveriam ter sido removidos. Esse processo de remoção total com subsequente reinserção de todos os dados pode ter um custo bastante elevado, principalmente quando se deseja manter boa parte dos valores já armazenados. Devido a essa característica, a estrutura Nanocubes não costuma ser utilizada com dados coletados continuamente e sim com um conjuntos limitados de dados "históricos", chamados de "datasets", porque o armazenamento contínuo de dados pode levar ao esgotamento da memória RAM disponível para armazená-los.

Trabalhos derivados da estrutura Nanocubes

Ao longo do tempo, foram desenvolvidas variantes Nanocubes cuja principal característica baseava-se na modificação do conteúdo dos valores armazenados nos vértices sumários (que no caso do Nanocubes eram contadores de ocorrências), reaproveitando a capacidade de organizar e compartilhar parte da estrutura e dos valores agregados para economizar memória. A estrutura TopKube [51] é uma dessas estruturas, onde o conteúdo dos valores armazenados nos vértices sumários é modificado para uma estrutura de dados apropriada para realizar ranqueamentos dos dados armazenados utilizando um algoritmo tipo Top-K [31]. Como resultado, a estrutura é capaz de responder rapidamente a consultas onde a informação desejada envolve a quantidade de ocorrências de algum evento. Como exemplo, os jogadores que mais fizeram pontos à partir de uma determinada região do campo de jogo e num determinado período de tempo.

Outra estrutura cujo elemento distintivo baseia-se na modificação dos valores armazenados nos vértices sumários é a estrutura Gaussiancubes [73]. Esta estrutura armazena valores necessários para produção de informações estatísticas mais complexas do que as produzidas pelo Nanocubes, como médias, método dos mínimos quadrados, PCA (Principal Component Analisys), etc. Uma contribuição importante desta estrutura é a utilização do armazenamento de valores nos vértices sumários que não são efetivamente a informação desejada, como no caso do Nanocubes.

Recentemente, uma tese de doutorado apresentou a estrutura Topocubes [71], cujos vértices sumários armazenam valores estruturados ("Boundary Matrix") para realização de **Topological Data Analysis**, uma técnica para análise multidimensional de dados, capaz de utilizar dados incompletos ou que incluem "ruído", como dados medidos em cosmologia, ciência dos materiais, etc. Da mesma forma que Gaussiancubes, a informação a ser retornada por um Topocubes depende de um processamento no momento da geração de resposta da consulta.

Todas estas estruturas, apesar de apresentarem variações no conteúdo dos vértices sumários, têm em comum uma certa especialização rígida, que impede, por exemplo, uma estrutura Gaussiancubes [73] armazenar informações que um TopKube [31] é capaz de armazenar. Essa falta de flexibilidade pode criar limitações operacionais importantes aos tomadores de decisão, uma vez que, ao selecionar uma dessas estruturas para gerenciar os dados, está impondo limites ao tipo de informação que poderá extrair. Para contornar essa limitação, torna-se necessário utilizar mais de uma estrutura ao mesmo tempo, o que exige uma múltipla carga de um grande volume de dados, que é claramente ineficiente.

Trabalhos afins com a estrutura Nanocubes

A estrutura de dados Hashedcubes [56] também é baseada no conceito básico de datacube, porém possui duas importantes diferenças em relação ao Nanocubes, ambas decorrentes de um esforço para reduzir a utilização de memória. A primeira delas implica em uma organização interna completamente diferente. O Hashedcubes não utiliza uma estrutura de dados similar a uma árvore, usando arestas (ponteiros) para organizar os dados, e sim um array multi-camadas, ordenado conforme os critérios definidos pelo utilizador da estrutura. Como é baseada em um array, o algoritmo de inserção de dados na estrutura foi concebido considerando que todos os dados a serem inseridos poderiam ser analisados antes de que qualquer inserção fosse realizada, ou seja, todos os dados têm de ser conhecidos a priori. Isso cria grandes dificuldades para inserção dinâmica, o que torna praticamente impossível utilizá-la com dados gerados continuamente. A segunda diferença é que o Hashedcubes não pré-agrega e armazena todas as combinações possíveis de consulta sobre os dados, como faz o Nanocubes. Isso reduz a utilização de memória, mas aumenta o tempo para os tipos de consulta não pré-agregados, pois pode exigir a agregação "on-th-fly" dos valores armazenados para produção das informações necessárias na resposta.

A estrutura ConcaveCubes [35] é estruturalmente baseada em Hashedcubes, oferecendo como diferencial, que são respostas contendo informações baseadas em "clusters

espaciais". Como um polígono, um "cluster espacial" delimita uma área de interesse calculada como resultado de uma consulta. Por exemplo, pode-se consultar uma estrutura ConcaveCubes, devidamente configurada, para descobrir quais os polígonos (clusters especiais) que determinam os limites das áreas onde existem casas à venda com 3 quartos e 2 banheiros. Como contribuição adicional, o artigo apresenta um novo algoritmo do tipo "Concave Hull Algorithm", que é capaz de criar polígonos côncavos (daí o nome da estrutura) com áreas reduzidas para implementação dos clusters especiais.

A estrutura Neuralcubes [72] se propõe a responder consultas com informações tipicamente originadas por datacubes, à partir do resultado calculado por redes neurais devidamente treinadas, ou seja, sem a utilização de datacubes e seus valores armazenados. Isso elimina a necessidade de conexão com algum tipo de gerenciador de dados para realização de exploração visual e interativa. No entanto, por ser um método preditivo, sabe-se que as informações produzidas não serão exatas, e portanto a análise desta tecnologia está fora do escopo deste trabalho por violar o requisito essencial *Req2*.

Recentemente, a estrutura Smartcube [39] apresentou uma nova abordagem para reduzir a utilização de memória em datacubes, contendo dados espaço-temporais multidimensionais. A estratégia do Smartcube consiste em computar inicialmente algumas partes do datacube, deixando para computar outras, à medida em que as consultas forem demandando. A ideia que norteia a abordagem é que existe uma certa focalização das consultas e, por conta disso, nem todas as informações pré-computadas num Nanocubes precisam ser de fato materializadas. Os resultados apresentados no trabalho mostram, como era de se esperar, uma redução significativa do uso da memória.

3.3 Considerações finais

Nenhum dos trabalhos apresentados consegue atender a todos os requisitos apresentados na Seção 1.1 a solução proposta se propõe a atender. Várias abordagens abrem mão da precisão das informações recuperadas para ganhar velocidade nas respostas, violando o requisito *Req2*. Outras lidam com grandes volumes de dados e produzem informações precisas, mas necessitam de recursos de vários computadores, o que aumenta o tempo de resposta, violando o requisito *Req4*. As estruturas de dados especializadas para lidar com EVID conseguem atender aos requisitos essenciais para o objetivo deste trabalho, porém tem dificuldades em lidar com dados realmente complexos, que exigem atender aos requisitos *Req5*, *Req6* e *Req7*.

A estrutura Nanocubes atende a diversos requisitos considerados necessários para atingir aos objetivos deste trabalho. Ela foi desenvolvida especificamente para possibilitar o recebimento de grandes volumes de dados complexos[*Req3*] (multidimensionais[*Req5*], espaciais[*Req6*] e temporais[*Req7*]), respondendo rapidamente[*Req4*] a consultas que demandam informações [*Req2*] baseadas nos dados recebidos[*Req1*]. Devido ao reaproveitamento interno de partes da estrutura, que evita o consumo desnecessário de memória, ela pode ser considerada eficiente [*Req14*]. Além disso, de acordo com os experimentos apresentados no artigo que define a estrutura, ele pode operar com grandes volumes de dados, utilizando equipamentos de uso cotidiano, o que a torna econômica[*Req15*]. Outro requisito atendido é a capacidade de interagir de forma padronizada com outros sistemas [*Req13*].

Por outro lado, existem alguns requisitos importantes que a estrutura não atende. Ela não é capaz de receber novos dados uma vez que já tenha atendido alguma consulta[*Req8*]. Ela também não oferece nenhum mecanismo para remoção de dados da estrutura, o que desrespeita os requisitos *Req9* e *Req10*. Além disso, a existência de trabalhos derivados do Nanocubes, que basicamente são modificações no tipo de informação produzida, demostra que a estrutura não é capaz de atender ao requisito *Req12*, que prevê possibilidade de utilização de novos tipos de informações.

O Nanocubes também apresenta limitações com relação aos tipos de dados com que é capaz de lidar, violando o requisito *Req11*. A estrutura está limitada à dados geográficos, categóricos e temporais, que tem de ser dispostos na estrutura exatamente nesta ordem. Não é possível utilizar dados com coordenadas tridimensionais, ou dados cuja semântica seja dada por novas funções e operações de consulta. Por exemplo, um novo tipo de dado pode definir um construtor que codifica os bits a serem armazenados na estrutura de uma forma especial para que possam ser utilizados por uma nova operação denominada "similar", que avalia, segundo critérios próprios, se dois dados são similares ou não.

Finalmente, não é possível garantir que o Nanocubes é a solução mais eficiente disponível sem a realização de uma análise mais aprofundada. De fato, antecipando alguns dos resultados deste trabalho, pode-se afirmar que o Nanocubes não possui os algoritmos mais eficientes com relação a economia de memória e, possivelmente, também em relação a tempo de execução.

Nos próximos capítulos serão apresentadas análises mais profundas da eficiência desta solução, considerada o estado da arte, e nas seções seguintes é apresentada a proposta de uma solução para todos os requisitos listados, inclusive aqueles limitantes do Nanocubes.

Capítulo 4

Nanocubes - Análises de eficiência

Este capítulo apresenta os resultados das avaliações da estrutura Nanocubes. A primeira seção deste capítulo apresenta uma releitura dos conceitos e regras utilizados em um Nanocubes de modo a facilitar a compreensão das análises de eficiência. Na seção seguinte é realizada uma análise da organização interna da estrutura, avaliando oportunidades para melhora de eficiência estrutural. Logo a seguir, apresenta-se a análise do algoritmo de inserção. Ao final, estão dispostas as conclusões decorrentes das análises realizadas.

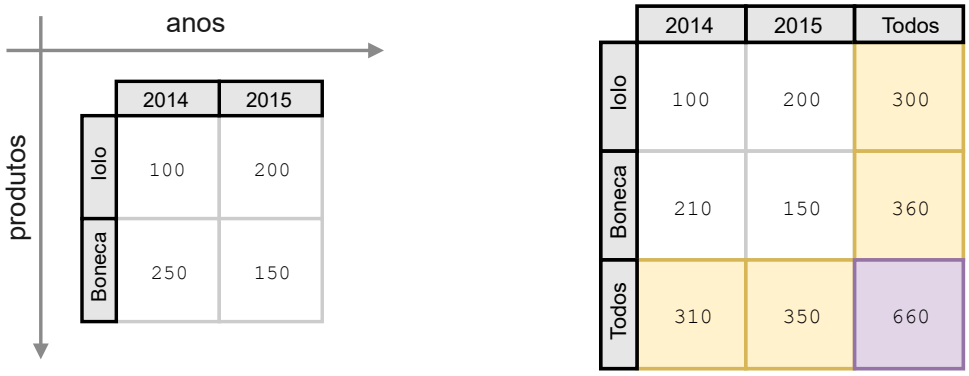
4.1 Introdução ao Nanocubes

Esta seção apresenta uma descrição da organização interna da estrutura de dados Nanocubes [38] evidenciando como um datacube é mapeado em um Nanocube, as estratégias utilizadas para reduzir o consumo de memória e manter o tempo de resposta (latência) baixo e também apresentando as regras e propriedades que todo Nanocube deve atender. Essa descrição tem como objetivo permitir a compreensão da organização e do funcionamento do Nanocubes para dar suporte à análise de eficiência, porém não adota exatamente a mesma abordagem utilizada no artigo que o descreve [38], que sempre será a referência mais apropriada para um estudo mais profundo dessa estrutura.

4.1.1 Visão geral

Nesta seção serão utilizados alguns exemplos para deixar o funcionamento da estrutura mais evidente. O primeiro exemplo na Seção 2.1 discorreu sobre um datacube bidimensional que possibilitava o rápido acesso a informações estatísticas. O exemplo a seguir também utiliza um datacube bidimensional, que é apresentado na forma de tabela.

Inicialmente considere a tabela da Figura 4.1 (a), onde as linhas identificam um produto, as colunas identificam um ano e o elemento localizado pela coordenada (produto; ano) é a quantidade daquele produto que foi vendido naquele ano. Com um pequeno esforço de abstração, pode-se considerar que a linha e a coluna que localizam os elementos na tabela são dois eixos de um plano cartesiano, onde cada lado do plano contém elementos de um conjunto-domínio formado por valores discretos.



(a) Essencial (b) Pré-agregações

Figura 4.1: Datacubes bidimensionais

A Figura 4.1(b) contém uma versão mais elaborada da tabela em (a), onde foram adicionadas linhas e colunas com o valor "Todos", também chamado de "Todos". Diferentemente dos demais valores, o valor "Todos" não existe realmente como um produto ou ano, ou seja, esse valor não existe como um elemento do domínio. O valor "Todos" é utilizado para indicar que a linha ou coluna contém a combinação agregada de todos os elementos "colaterais" desse valor. Por exemplo, quando "Todos" aparece numa coluna, então o valor nessa coluna em cada linha corresponderá ao valor agregado de todos os outros valores possíveis das outras colunas na linha. Da mesma forma, quando "Todos" aparece numa linha, agregam-se todos os elementos das outras linhas para cada coluna. Na Figura 4.1(b), vê-se esses valores agregados à partir de valores reais com a cor de fundo amarela. Já o elemento com a cor de fundo lilás, é uma agregação dos valores "Todos" já agregados nas linhas e colunas, porém é consistentemente definido, seja considerado como elemento das linhas ou das colunas.

A Figura 4.2 apresenta uma versão didática de um possível nanocube[1] que armazena a mesma informação que a tabela na Figura 4.1(b). Inicialmente considere apenas os textos ao lado das linhas desenhadas denominadas arestas. Observe que no primeiro "nível" de texto estão os valores correspondentes às linhas dos produtos. Já no segundo "nível" de

[1] um nanocube é uma instância da estrutura Nanocubes

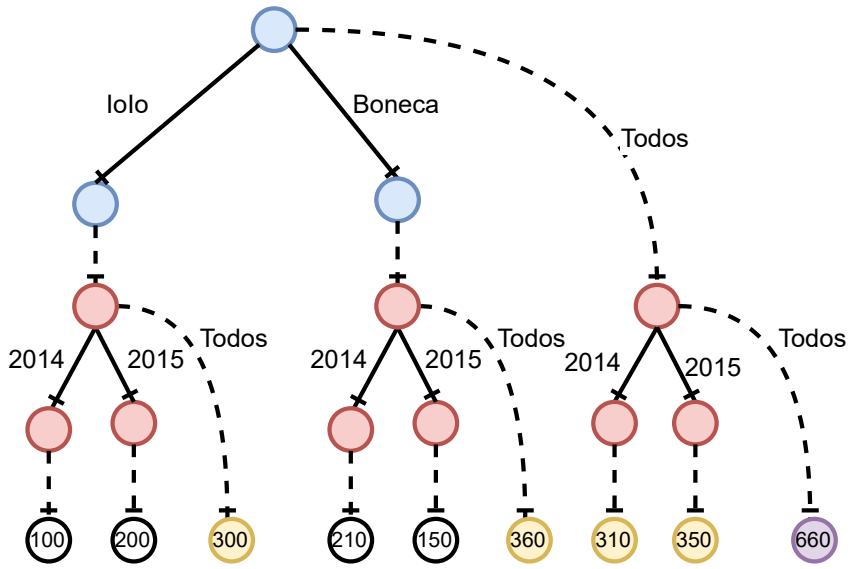


Figura 4.2: Representação didática de um nanocube para tabela na Figura 4.1(b)

texto, estão os valores correspondentes às colunas dos anos. Note que, em ambos os casos, o valor "Todos" está presente. Já a última linha de texto contém círculos, doravante chamados de vértices, com números. Observe os valores em cada vértice (círculo) com fundo branco correspondem a algum elemento da tabela. Já os valores em círculos amarelos correspondem aos valores nas linhas e colunas "Todos". Finalmente, o círculo com fundo lilás tem um valor idêntico ao elemento com fundo lilás na tabela. Além disso, observe que os vértices nas extremidades de arestas com nomes referentes aos produtos tem a cor azul e os associados a arestas referentes aos anos têm cor vermelha. Isso é intencional e indica visualmente que diferentes dimensões dos valores estão representadas com cores distintas. Note que, por questões didáticas, os círculos na última linha não possuem a mesma cor. As cores desses círculos ajudarão a evidenciar mais informações presentes nesse diagrama.

A Figura 4.2 também revela importantes detalhes com respeito aos elementos da tabela. Partindo do círculo mais alto na figura pode-se notar que a sequência de arestas identificadas é uma versão sequencial do sistema de coordenadas da tabela. Por exemplo, partindo do vértice no topo e seguindo pela aresta "Ioio", uma aresta sem texto e depois a aresta "2015", chega-se a um círculo com o valor "200", que corresponde ao valor na coordenada (IoIo; 2015) da tabela. Alternativamente, partindo do vértice no topo e seguindo pela aresta "Todos" e depois pela aresta "2015", chega-se a um círculo com o valor "350" que corresponde ao valor na coordenada (Todos; 2015). Ao se percorrer exaustivamente todos os caminhos enquanto se registra os nomes das arestas, pode-se constatar que todos os valores armazenados na tabela podem ser encontrados utilizando

a representação oferecida por este nanocube.

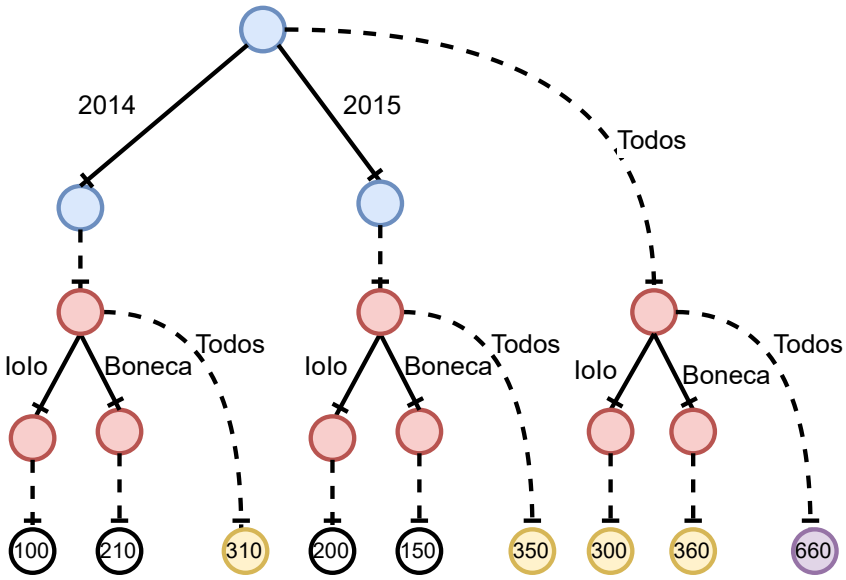


Figura 4.3: Nanocube alternativo para a tabela na Figura 4.1(b)

Já a Figura 4.3 apresenta outra abordagem para representar a mesma tabela utilizando Nanocubes. A diferença em relação à Figura 4.2 é que a ordem em que as dimensões foram dispostas no nanocube está invertida. Na figura anterior, a ordem das dimensões à partir do vértice inicial era "produto" seguido de "ano". Na figura atual, a ordem é "ano" e depois "produto". Observe que não há perda de informação, apenas mudança na disposição dos totais na última linha.

	2014			2015			Todos (C)
	1º Sem	2º Sem	Todos (A)	1º Sem	2º Sem	Todos (B)	
Iolo	40	60	100	120	80	200	300
Boneca	80	130	210	110	40	150	360
Todos (D)	120	190	310	230	120	350	660

Figura 4.4: Datacube apresentando o tempo como uma hierárquia

Datacubes devem ajudar no processo de exploração das informações oriundas de dados. Por vezes, é interessante diminuir a quantidade da informação sendo exibida para não sobrecarregar o usuário do datacube e permitir uma visão mais geral. Porém, também é interessante possibilitar que o usuário "mergulhe" nos detalhes da informação que ele tem interesse. Para atender a essa necessidade, os datacubes permitem que os valores correspondentes às dimensões sejam agrupados em níveis hierárquicos. Utilizando hierarquias,

o usuário do datacubes pode descer de um nível de informação mais alto, que permite uma visão geral, até um nível mais baixo, que oferece mais detalhes. O caminho inverso, do detalhe para o geral, também pode ser percorrido.

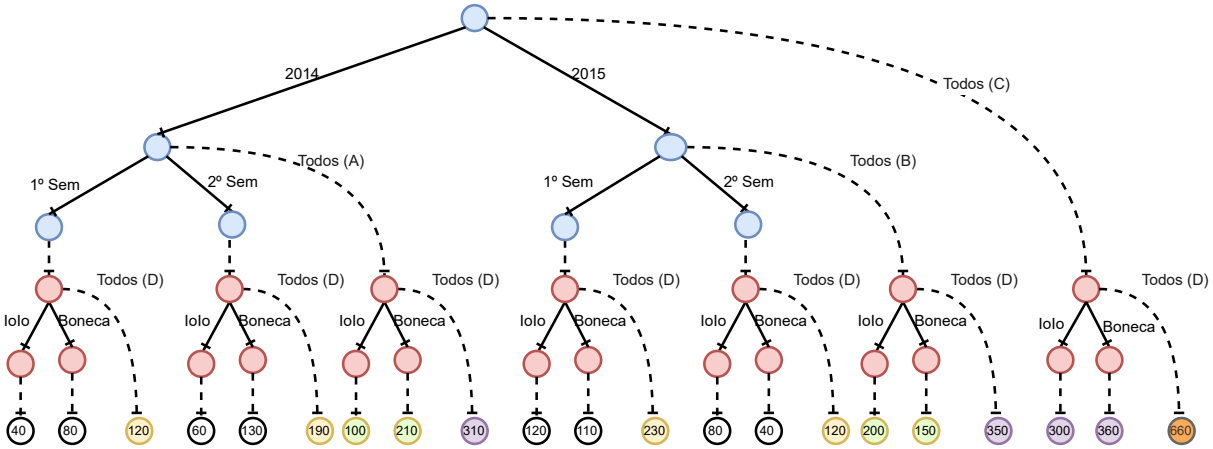


Figura 4.5: Nanocube com hierarquia de tempo para datacube da Figura 4.4

A Figura 4.4 exibe um datacube bidimensional onde a dimensão de tempo é definida hierarquicamente. Nesse datacube, o nível mais elevado da dimensão tempo é composto por "anos" e o nível com mais detalhes contém "semestres". Observe que, com a criação de apenas uma hierarquia para dimensão de tempo, o número de colunas "Todos" aumentou de um para três. Isso ocorre porque agora cada ano tem sua própria coluna "Todos", que deve armazenar a agregação dos valores dos semestres do ano. Como são dois anos, são necessárias mais duas colunas "Todos". Por razões didáticas, as colunas (linhas) "Todos" receberam um identificador que permite identificar em qual posição do datacube a coluna (ou linha) "Todos" está.

A Figura 4.5 exibe uma representação didática de um nanocube correspondente ao datacube da Figura 4.4, com as dimensões apresentadas na ordem "ano / semestre" → "produto". Observe a correspondência da coluna (linha) "Todos" no datacube com a posição da aresta "Todos" no nanocube. Note também a relação entre os totais do datacube e do nanocube. Por exemplo, a coluna "Todos (C)" tem como correspondente a aresta com o mesmo nome no nanocube. Assim, ao se percorrer a arestas "Todos (C)" tem-se 3 opções: seguir pela aresta "Todos (D)" e chegar ao grande total do datacube ou seguir por uma das duas arestas de produto, que levarão aos respectivos subtotais de cada produto. Ao mesmo tempo, seguindo por uma aresta de ano, por exemplo 2014, tem-se 3 opções. Uma delas é seguir pela aresta "Todos" para 2014, que levará diretamente para totalizações do ano, agrupando todos os semestres. As demais opções levam ao próximo nível de detalhamento, que no caso são os semestres do ano. Ao se escolher uma delas, acessa-se a

dimensão de produto, onde é possível descobrir o total para todos os produtos para aquele semestre do ano, ou apenas o total de um dos produto para o mesmo semestre do ano.

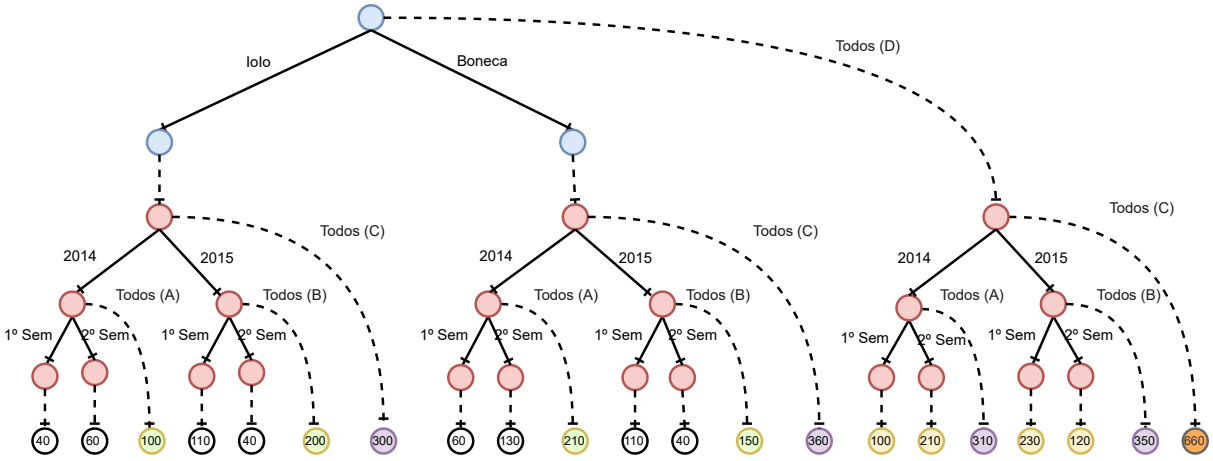


Figura 4.6: Datacube da Figura 4.4 na ordem "produto" → "ano / semestre"

A Figura 4.6 também exibe um nanocube correspondente ao datacube da Figura 4.4. No entanto, diferentemente do nanocube da Figura 4.5, este nanocube tem uma ordem diferente das dimensões, colocando a dimensão "produto" mais próxima do vértice inicial do que a dimensão "ano/semestre". Não existe nenhuma perda de informação com essa mudança e o número de vértices que armazenam valores é o mesmo, porém o número total de vértices é diferente nos dois nanocubes. No primeiro nanocube exibido, existem 49 vértices e no último, apenas 45. Isso acontece porque, uma vez descontados os 21 vértices correspondentes aos elementos da tabela, o restante da estrutura é montado de forma diferente. No primeiro nanocube, existem 7 grupos de 3 vértices na segunda dimensão (21 vértices) e 7 vértices na primeira, levando o total para 28 vértices. No segundo, existem 3 grupos de 7 elementos na segunda dimensão (21 vértices), mas apenas 3 vértices na primeira dimensão, levando o total para 24 vértices. Este exemplo demonstra que a ordem em que as dimensões são organizadas pode causar impacto na quantidade de recursos alocados para um nanocube.

4.1.2 Conceitos, definições e propriedades

A Figura 4.7 apresenta os nomes utilizados pelo Nanocubes para identificar seus elementos, bem como as convenções de cores utilizadas nesta seção para representação de um nanocube. Inicialmente, pode-se perceber que os vértices da última linha agora têm apenas a cor verde. Consultando à legenda, pode-se entender o esquema de cores utilizado para identificar as diferentes dimensões dentro de um Nanocubes. Nota-se também que os únicos vértices que armazenam os valores dos elementos na tabela são os vértices da

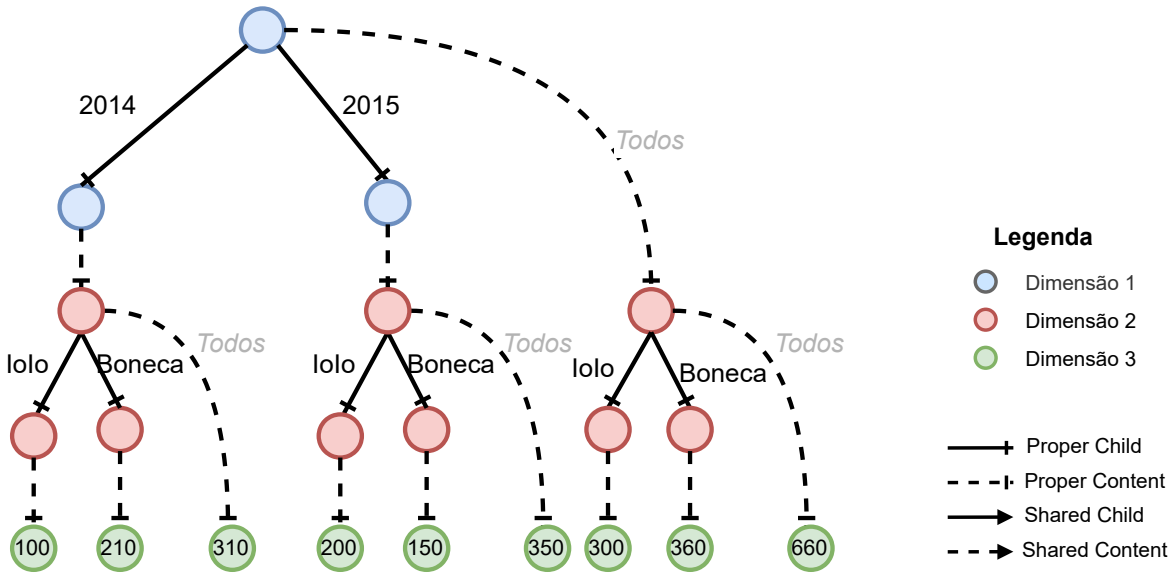


Figura 4.7: Representação do nanocube da Figura 4.3 (quase) no formato final

maior dimensão do nanocube. A ideia é que o caminho entre o vértice inicial e o vértice na dimensão mais alta coincida com a coordenada do elemento na tabela, que, de fato, é um datacube (bidimensional). Esse vértices de dimensão mais alta são denominados "**Sumários**" do inglês "**Summary**"

Na Figura 4.7 destaca-se que o nome "Todos", que aparece ao lado de algumas arestas pontilhadas, agora está quase invisível. Isso acontece porque, na realidade, o Nanocubes não registra explicitamente se uma aresta **Proper Content** está representando o valor "Todos" em um datacube ou não. Essa anotação ao lado da aresta só está presente para facilitar a compreensão. Observe que todas as arestas tracejadas (**Content**) só ocorrem entre elementos com cores (dimensões) diferentes. Isso acontece porque a função primária das arestas **Content** em um Nanocubes é interligar dimensões. Uma aresta **Proper Content** representa o valor "Todos" quando tem origem em um vértice que tem uma ou mais arestas **Proper Child** ou **Shared Child** porque só faz sentido falar em "Todos" quando existe pelo menos algum valor a se considerar. Assim, para o Nanocubes, quando uma aresta **Proper Content** tem origem num vértice sem outras arestas, ela não está representando o valor "Todos"; a aresta está apenas interligando vértices que estão em dimensões diferentes.

Nessa figura, ainda é possível observar que as arestas **Proper Child** sempre estão ligando vértices da mesma cor. Isso também é intencional e ocorre porque, no Nanocubes, a função principal das arestas **Child** é definir a estrutura interna das dimensões.

Neste ponto já é possível relacionar algumas propriedades reconhecidas do Nanocubes:

Nn1 Os vértices sumários são os únicos que armazenam valores, que correspondem a agregações;

Nn2 Por representarem agregações, os vértices sumários não possuem arestas;

Nn3 As arestas **Child** são as únicas que possuem valores, que são originados dos valores encontrados nas dimensões correspondentes no datacube;

Nn4 As arestas **Content** não possuem valores, podendo estar associadas ao valor "Todos" ou não;

Nn5 Vértices que estão em dimensões diferentes tem de ser interligados através de arestas **Content**;

Nn6 Quando uma aresta **Proper Content** tem origem num vértice sem outras arestas, então ela apenas está interligando duas dimensões;

Nn7 Quando uma aresta **Proper Content** tem origem num vértice com duas ou mais arestas, então esta aresta está representando o valor "Todos" de um datacube;

Nn8 Vértices que estão na mesma dimensão tem de ser interligados por arestas **Child**;

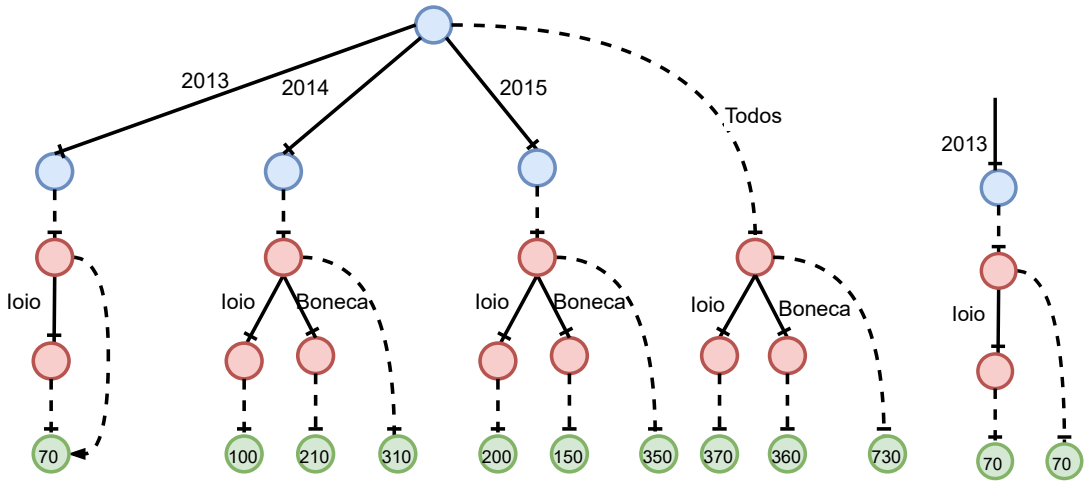
Pode-se notar, na lista de propriedades, a falta da informação sobre uma aresta **Proper** no caso em que um vértice é origem de apenas uma aresta **Child**. O próximo exemplo apresentará a abordagem do Nanocubes para esta situação.

A Figura 4.8(a) mostra um datacube bidimensional refletindo um cenário no qual o administrador de uma empresa decidiu inserir os dados de 2013, quando a linha de produção só fabricava ioios. A Figura 4.8(b) exibe um nanocube que representa esse datacube. O primeiro fato a se observar é que o Nanocubes não cria vértices ou arestas quando não existe informação a ser representada. No caso, como não havia produção de bonecas em 2013, diferentemente da ilustração do datacube que reserva espaço para a produção de bonecas, o Nanocubes não aloca nenhum vértice. Outro fato relevante é a utilização da aresta **Shared Content**, que aproveita o resultado agregado individual para representar o valor "Todos". A Figura4.8(c) ilustra que, na ausência de uma aresta do tipo **Shared Content**, seria necessário criar um novo vértice sumário apenas para acomodar uma agregação que é idêntica registrada no único valor na coluna 2013. Dito de outra forma o elemento na coordenada (2013; ioio) tem o mesmo valor que o elemento na coordenada (2013; "Todos"), porque "Todos" só inclui ioio.

A Figura 4.9(a) mostra outro datacube bidimensional refletindo um novo cenário no qual uma empresa mudou sua linha de produção em 2016, substituindo ioios por piões. A

	2013	2014	2015	Todos
Ioio	70	100	200	370
Boneca		210	150	360
Todos	70	310	350	730

(a) Datacube com o ano de 2013



(b) Nanocube correspondente (c) 2013

Figura 4.8: Exemplo de uso de aresta Shared Content

Figura 4.9(b) exibe um nanocube que representa esse novo datacube. Na figura |4.9(a), nota-se que o produto Pião só aparece em duas colunas do datacube, sendo uma delas a coluna "Todos". O Nanocubes aproveita-se desse fato para evitar a criação de vértices adicionais. Na Figura 4.9(b), percebe-se que a agregação obrigatória para "pião" em 2016 é utilizada, através de uma aresta **Shared Child**, para fornecer a agregação para ("Todos"; pião). Isso ocorre porque o elemento na coordenada (2016, pião) tem o mesmo valor que o elemento na coordenada ("Todos", pião) porque o único ano em que o produto "pião" existe é 2016.

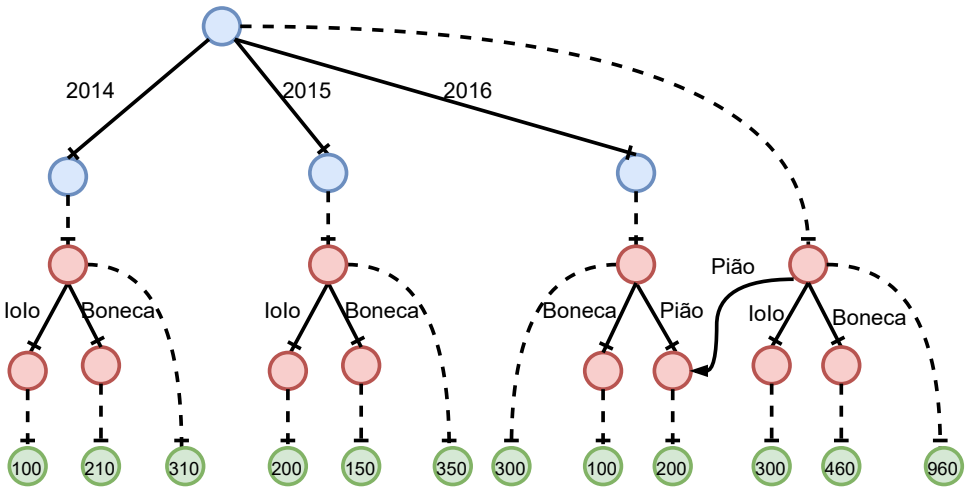
Em face das novas constatações, é possível apresentar mais algumas propriedades do Nanocubes que promovem economia de memória ou dão flexibilidade à estrutura:

Nn9 O Nanocubes não registra informações de elementos do produto cartesiano das dimensões do datacube que estejam ausentes;

Nn10 O Nanocubes utiliza a aresta **Shared Content** para sinalizar um valor "Todos" quando existe apenas uma outra aresta com valor no mesmo vértice, evitando a criação de vértices adicionais que serviriam apenas para armazenar um valor de

	2014	2015	2016	Todos
Ioio	100	200		300
Boneca	210	150	100	460
Pião			200	200
Todos	310	350	300	960

(a) Datacube com a troca da produção de ioio por pião em 2016



(b) Nanocube correspondente

Figura 4.9: Exemplo de uso de aresta Shared Child

agregação que é igual ao único elemento existente;

Nn11 O Nanocubes utiliza a aresta **Shared Child** para aproveitar o fato de que só existe um elemento e subordinado ao valor "Todos", o que garante que ("Todos", ...) = (e, ...). Assim, o Nanocubes aproveita, através da aresta **Shared Child**, todo o desenvolvimento estrutural do elemento e, evitando a duplicação de mais vértices e arestas;

Nn12 Todos os vértices de um Nanocubes, à exceção dos sumários, possuem uma aresta **Content**, tornando possível a existência de um caminho que permite mudar imediatamente de dimensão à partir de qualquer vértice.

4.2 Nanocubes - Organização e regras relevantes

A organização interna da estrutura Nanocubes já foi discutida na Seção 4.1, onde foram apresentadas algumas regras que um nanocube deve atender. De fato, parte da análise da estrutura foi realizada naquela seção, onde foram evidenciadas algumas regras importantes para construção da estrutura.

Neste capítulo focaremos na análise de algumas consequências das seguintes regras:

1. Vértices que estão em dimensões diferentes têm de ser interligados através de arestas **Content**;
2. Quando uma aresta **Proper Content** tem origem num vértice sem outras arestas, então ela apenas está interligando duas dimensões;
3. Vértices que estão na mesma dimensão tem de ser interligados por arestas **Child**;

```
1: function NanoCube([o_1, o_2, ..., o_n], S, ℓ_time)          ▷ n > 0
2:     nano_cube ← Node()                                      ▷ New empty node
3:     for i = 1 to n do
4:         updated_nodes ← ∅
5:         Add(nano_cube, o_i, 1, S, ℓ_time, updated_nodes)
6:     end for
7:     return nano_cube
8: end function

1: function TrailProperPath(root, [v_1, ..., v_k])
2:     stack ← Stack()                                         ▷ New Empty Stack
3:     Push(stack, root)
4:     node ← root
5:     for i = 1 to k do
6:         child ← Child(node, v_i)
7:         if child = null then
8:             child ← NewProperChild(node, v_i, Node())
9:         else if IsSharedChild(node,child) then
10:            child ← ReplaceChild(node,
                   child, ShallowCopy(child))
11:        end if
12:        Push(stack, child)
13:        node ← child
14:    end for
15:    return stack
16: end function

1: function ShallowCopy(node)
2:     node_sc← Node()
3:     SetSharedContent(node_sc, Content(node))
4:     for v in ChildrenLabels(node) do
5:         NewSharedChild(node_sc, v, Child(node, v))
6:     end for
7:     return node_sc
8: end function
```

```
1: procedure Add(root, o, d, S, ℓ_time, updated_nodes)
2:     [ℓ_1, ..., ℓ_k] ← Chain(S, d)
3:     stack ← TrailProperPath(root, [ℓ_1(o), ..., ℓ_k(o)])
4:     child ← null
5:     while stack is not empty do
6:         node ← Pop(stack)
7:         update ← false
8:         if node has a single child then
9:             SetSharedContent(node, Content(child))
10:        else if Content(node) is null then
11:            SetProperContent(node,
                   ( d= dim(S) ?
                       SummedTableTimeSeries() : Node() )
12:            update ← true
13:        else if ContentIsShared(node) and
                   Content(node) not in updated_nodes then
14:            SetProperContent(node,
                   ShallowCopy(Content(node)))
15:            update ← true
16:        else if ContentIsProper(node) then
17:            update ← true
18:        end if
19:        if update then
20:            if d= dim(S) then
21:                Insert(Content(node), ℓ_time (o))
22:            else
23:                Add(Content(node), o, d+1, updated_nodes)
24:            end if
25:            Insert(updated_nodes, Content(node))
26:        end if
27:        child ← node
28:    end while
29: end procedure
```

Figura 4.10: Algoritmo de inserção do Nanocubes
Fonte: Nanocubes [38]

Complementando a informação já apresentada, o artigo do Nanocubes [38] define formalmente um nanocube utilizando o conceito de *Chain*, um mecanismo através do qual

se obtém os valores para cada aresta da estrutura, indiretamente definindo os possíveis caminhos que podem existir. O fato a se destacar, é que cada dimensão possui um *Chain* próprio, e o algoritmo de inserção do Nanocubes utiliza este fato para organizar toda a estrutura.

4.3 Análise da estrutura Nanocubes

A atividade de pesquisa *AP1* apresentada na Seção 1.2, determina a análise do estado da arte tentando identificar oportunidades para melhorar a eficiência na utilização de recursos. Caso seja encontrada alguma oportunidade de melhoria, deve-se investigar a viabilidade da criação de uma nova estrutura que faça melhor aproveitamento dos recursos computacionais.

4.3.1 Análise

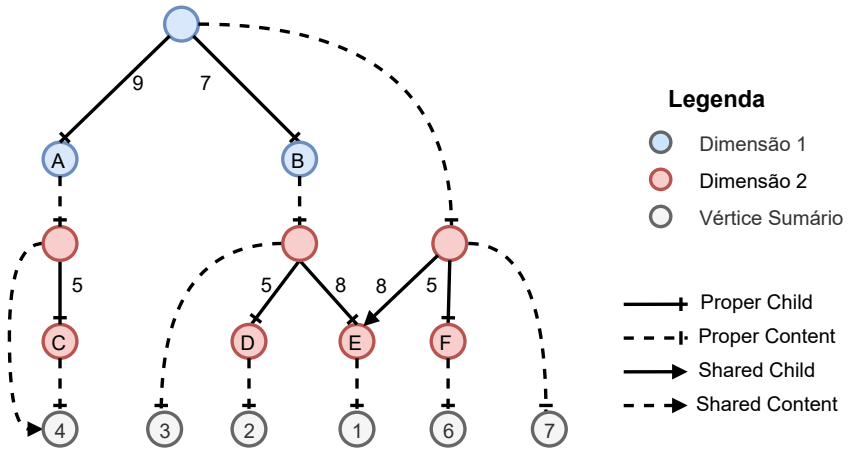


Figura 4.11: Nanocube identificando vértices de ligação

A Figura 4.11 apresenta um nanocube que será utilizado neste processo de análise. A consequência direta da regra 2 é a criação de vértices de ligação, que só existem para interligar dimensões através de arestas **Content**, como os vértices identificados como A, B, C, D, E e F na Figura 4.11. Isso ocorre porque o algoritmo de inserção do Nanocubes manipula a estrutura por partes, cada qual correspondente a uma dimensão. O algoritmo progride através de arestas **Child** dentro de uma dimensão até encontrar um vértice que não possui aresta **Child**. Neste caso, o algoritmo "entende" que é necessário mudar de dimensão, e usa uma aresta **Proper Content** para isso.

Como os vértices de ligação não são origem de nenhuma aresta do tipo **Child**, não

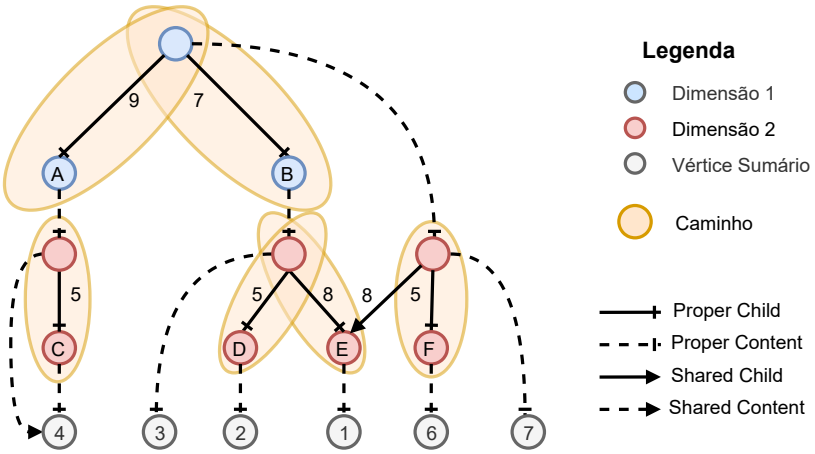


Figura 4.12: Caminhos entre dimensões em um nanocube

auxiliam no processo de organização da informação da mesma forma que os demais vértices. Além disso, como todos os vértices não sumários em um Nanocubes, esses vértices de ligação têm de possuir uma aresta **Content**, o que, de alguma forma, demanda mais utilização de memória.

4.3.2 Estrutura alternativa

A Figura 4.13 apresenta uma instância de uma estrutura de dados diferente, que preserva diversas características de vértices e arestas dos Nanocubes, mas elimina todos os vértices de ligação. Do ponto de vista da utilização de recursos, essa alteração diminui o consumo de memória, tornando a solução potencialmente mais eficiente[2].

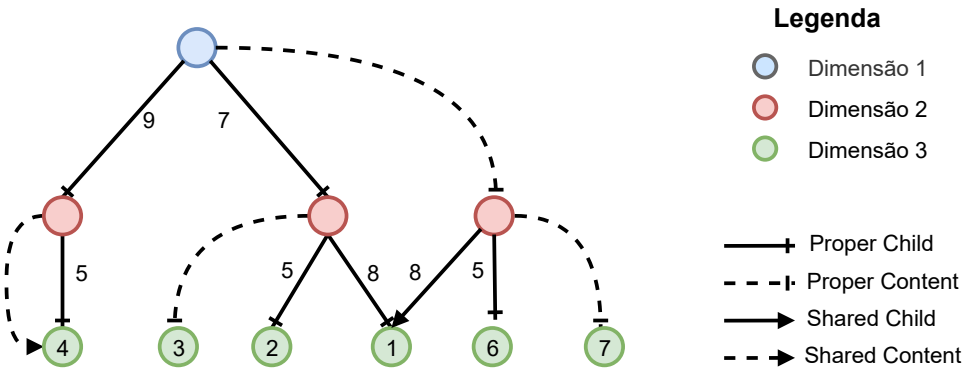


Figura 4.13: Instância da estrutura de dados proposta equivalente ao nanocube anterior

Por outro lado, a eliminação de vértices de ligação nessa nova estrutura implica na revogação da regra que impede arestas do tipo **Child** de interligarem vértices de diferentes

[2]a eficiência pode levar em consideração outros fatores, como o impacto no tempo de execução.

dimensões. Ao se revogar essa regra, a definição de *Chain* do Nanocubes não é mais consistente. E sem o conceito de *Chain*, elemento que indiretamente define os caminhos a se percorrer dentro de uma dimensão, o algoritmo de inserção do Nanocubes (algoritmo "ADD") não pode mais ser utilizado na nova estrutura.

4.3.3 Considerações

A obrigatoriedade no uso de vértices de ligação na estrutura Nanocubes pode ser considerada perda de eficiência na utilização de recursos e, portanto, caracteriza-se como uma resposta para a atividade de pesquisa correspondente a análise da estrutura Nanocubes com relação a eficiência (atividade *AP1*).

A eliminação dos vértices de ligação implica na criação de uma nova estrutura de dados que, embora possa utilizar diversas características da estrutura Nanocubes, não pode usar o seu algoritmo de inserção devido a ausência do conceito de *Chain*. Por conta desta incapacidade, para que se possa criar uma estrutura mais eficiente baseada no Nanocubes, será necessário desenvolver um novo algoritmo de inserção.

4.4 Análise do algoritmo de inserção

O algoritmo de inserção do Nanocubes, listado na Figura 4.10, é responsável por criar e manter instâncias do Nanocubes que estejam de acordo com as propriedades definidas para a estrutura. Esta seção apresenta uma análise do algoritmo de inserção, com ênfase na detecção de oportunidades para melhoria de eficiência.

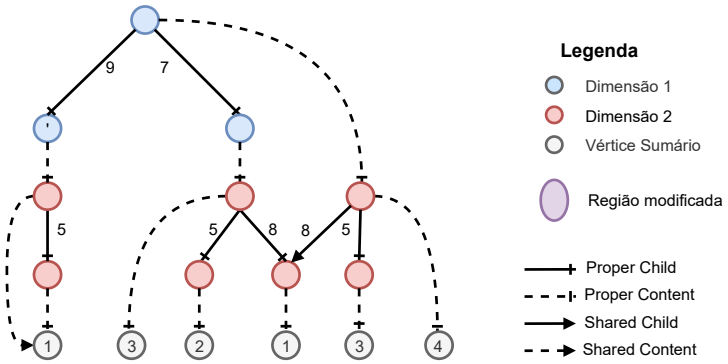


Figura 4.14: Nanocube após algumas inserções

Resumidamente, o algoritmo de inserção, "procedure ADD", consiste em, à partir do vértice raiz, descer pelas dimensões percorrendo os caminhos definidos pelas *Chains*,

criando ou ajustando arestas **Child** ao longo desse caminho e chamando a rotina "TrailProperPath" para isso. Ao retornar da chamada de "TrailProperPath", o algoritmo deve ajustar as dimensões que foram atingidas através de arestas **Content** dos vértices visitados na descida, caso não tenham sido ajustadas antes.

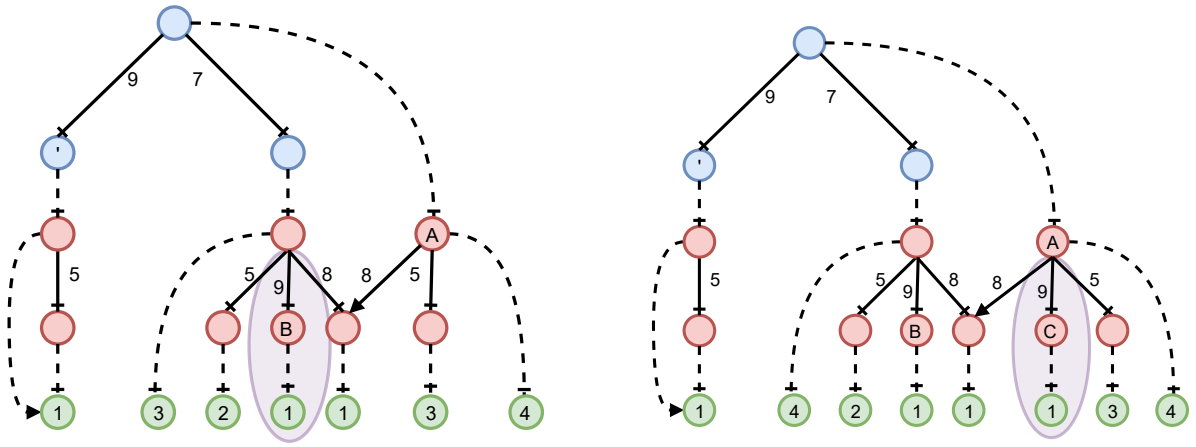
A rotina "TrailProperPath", que faz a maioria dos ajustes na estrutura, funciona da seguinte forma: à partir do inicio de uma chain, ela deve "descer" pelas arestas, usando os valores a serem armazenados como seletores de arestas, até chegar ao final da chain. Caso não exista uma aresta com o valor a ser armazenado, ela cria um novo vértice, cria uma aresta **Proper Child** para acessá-lo, armazena o valor nessa nova aresta e continua descendo à partir desse novo vértice criado. Caso exista uma aresta com o valor a ser armazenado, mas ela seja uma aresta **Shared Child** , então a rotina cria um novo vértice, o coloca como destino da aresta **Shared Child** original, mas transforma essa aresta em **Proper Child**. Além disso, a rotina chama a rotina "ShallowCopy", que copiará todas as arestas com origem no vértice antigo mas colocando o vértice recém criado como origem e fazendo com que todas as arestas copiadas sejam **Shared Child** ou **Shared Content**.

4.4.1 Análise

As análises nesta seção utilizam a Figura 4.14, que apresenta um nanocube com duas dimensões, onde cada dimensão contém apenas uma aresta **Child**. Assume-se que os valores nas arestas são obtidos de pares ordenados de números inteiros, com o primeiro valor do par definindo o valor da aresta da dimensão 1, e o segundo valor do par, o valor da aresta na dimensão 2. Para fins de análise, não é necessário detalhar como este nanocube atingiu este estado.

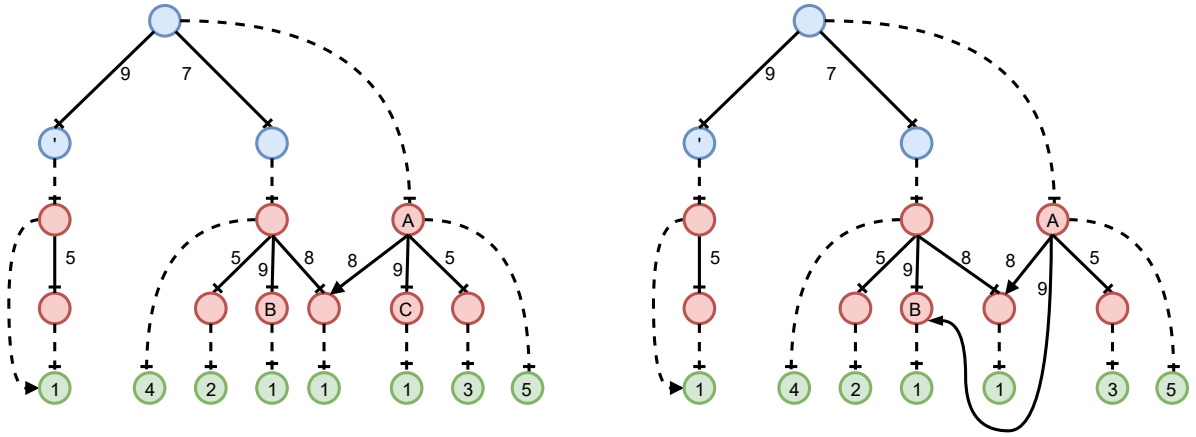
4.4.1.1 Peculiaridade 1: Valores inéditos

Considere que um par com valor (7,9) deve ser inserido no nanocube da Figura 4.14. A Figura 4.15 exibe etapas relevantes da transformação do nanocube durante a utilização do algoritmo de inserção. Na Figura 4.15(a), o algoritmo de inserção corretamente criou o vértice B e o vértice sumário descendente dele com o valor 1. Na Figura 4.15(b), contudo, quando a rotina "TrailProperPath" é executada usando o vértice A como raiz, ocorrem os seguintes eventos indesejáveis. Na linha 7, a rotina detecta que não localizou nenhuma aresta com valor 9, então cria uma nova aresta PROPER CHILD com este valor, ligando o vértice A e o novo vértice C (linha 8). A partir daí, o algoritmo de inserção progride da forma desejada até criar o vértice sumário descendente de C. A parte (a) da



(a) Novo caminho até vértice sumário (b) TrailProperPath - ReplaceChild

Figura 4.15: Etapas da inserção do par (7,9)



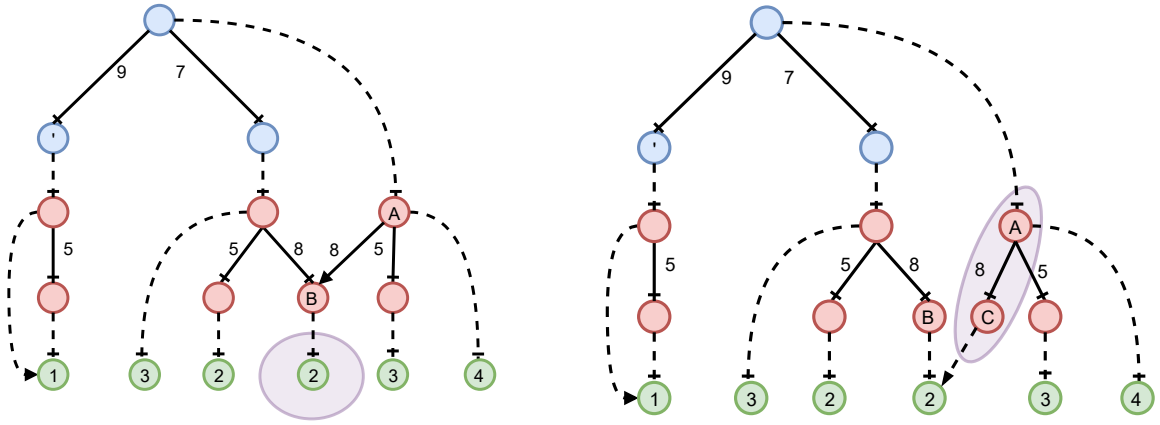
(a) Resultado final (b) Resultado desejado

Figura 4.16: Resultado final e desejado após inserção do par (7,9)

Figura 4.16 mostra a configuração final do nanocube que continua produzindo informações consistentes, porém com vértices e arestas adicionais desnecessários. Note que a aresta que aponta para o vértice B é a única descendente do vértice inicial que possui o valor 9. Isso permitiria que o vértice B fosse apontado pelo vértice A através de uma aresta SHARED CHILD. Ao contrário, o algoritmo criou uma aresta PROPER CHILD, um novo vértice C e o vértice sumário descendente de C. A parte (b) da mesma figura mostra o resultado ideal que o algoritmo de inserção deveria produzir.

4.4.1.2 Peculiaridade 2: Reinserção

Considere que um par com valor (7,8) deve ser inserido no nanocube da Figura 4.14. A Figura 4.17 exibe etapas relevantes da transformação do nanocube durante a execução

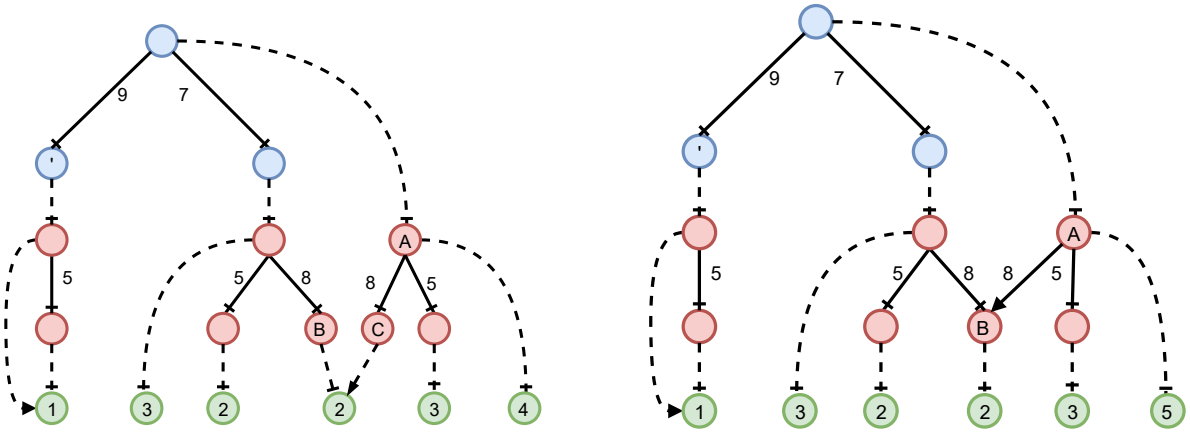


(a) Incremento do conteúdo do vértice terminal

(b) TrailProperPath - ReplaceChild

Figura 4.17: Etapas da inserção do par (7,8)

do algoritmo de inserção. Na parte (a) da figura, o algoritmo de inserção corretamente incrementou o total no vértice sumário, filho do vértice identificado como B. No entanto, quando a rotina "TrailProperPath", usando o vértice A como raiz, é executada, a linha 9 dessa rotina identifica a aresta com valor 8 como sendo **Shared Child** que aponta para o vértice B. Como resultado, o algoritmo executa a linha 10 que cria um novo vértice (vértice C) e substitui a aresta **Shared Child** por uma aresta **Proper Child** apontando para o vértice C, como é ilustrado na parte (b) da figura.



(a) Resultado Final

(b) Resultado Desejado

Figura 4.18: Resultado final e desejado após inserção do par (7,8)

A Figura 4.18 (a) mostra a configuração do nanocube após a inserção. O nanocube continua produzindo informações consistentes, porém com vértices e arestas adicionais desnecessários. A Figura 4.18 (b) exibe qual a configuração que nanocube poderia ter para não haver desperdício de recursos.

4.4.2 Considerações

O comportamento do algoritmo reduz a eficiência na utilização dos recursos ao criar arestas, vértices intermediários e um vértice sumário adicionais desnecessários. Note que, na estrutura Nanocubes, vértices sumários podem armazenar séries temporais com grande quantidade de dados, podendo utilizar uma quantidade significativa de memória. Note também que, no Nanocubes, cada novo vértice com mais do que uma aresta **Child**, necessariamente cria uma aresta **Proper Content** que provavelmente levará a criação de vários outros vértices, incluindo vértices terminais. Ao criar vértices desnecessários e duplicar vértices sumários desnecessariamente, o algoritmo se afasta do seu principal objetivo, que é minimizar o consumo de memória.

4.5 Conclusão

A análise da estrutura Nanocubes (Seção 4.3) identificou o potencial desperdício de recursos devido ao uso obrigatório de vértices que interligam caminhos entre duas dimensões diferentes (vértices de ligação). A estrutura alternativa proposta naquela seção solucionaria o problema do desperdício de recursos. No entanto, ainda é necessário descobrir se é possível desenvolver um algoritmo de inserção que não necessite de vértices de ligação e ainda seja capaz de preservar as demais propriedades do Nanocubes. A atividade de pesquisa *AP3* se propõe a resolver esta questão.

A análise do algoritmo de inserção do Nanocubes (Seção 4.3) levou a constatação de que, mesmo em situações cotidianas, o algoritmo pode deixar de realizar compartilhamento de partes da estrutura, que é uma das características que garantem a boa eficiência na utilização de recursos apresentada pelo Nanocubes.

Como consequência das análises realizadas, constatou-se que é adequado desenvolver uma nova uma estrutura de dados, que pode ser similar ao Nanocubes para aproveitar suas melhores características, porém deve evitar o desperdício causado por vértices de ligação. Essa nova estrutura deverá utilizar um novo algoritmo de inserção que seja capaz de preservar o compartilhamento promovido pelo Nanocubes sem apresentar as peculiaridades encontradas.

Capítulo 5

Tinycubes - Visão geral

A tecnologia modular Tinycubes foi desenvolvida para permitir a exploração visual e interativa de grandes volumes de dados espaço-temporais multidimensionais gerados continuamente. Este capítulo apresenta uma visão geral da tecnologia, descrevendo os elementos que a compõem e introduzindo os conceitos essenciais para sua compreensão.

5.1 Introdução

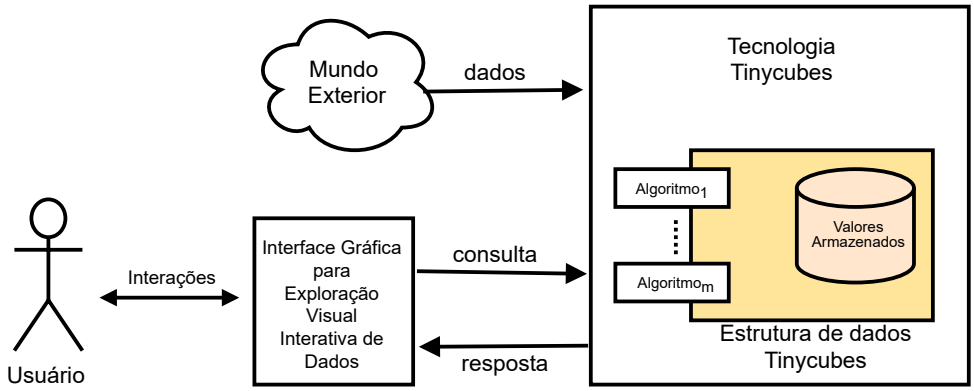


Figura 5.1: Esquema simplificado da Tecnologia Tinycubes

A Figura 5.1 ilustra a organização da tecnologia Tinycubes. O núcleo da tecnologia é a estrutura de dados Tinycubes, concebida para atender a consultas que demandam informações em tempo reduzido. Os algoritmos que implementam essa estrutura foram projetados para utilizar eficientemente os recursos computacionais disponíveis, em especial, a memória RAM. A estrutura foi planejada para ser modular, possibilitando que novos tipos de dados possam ser recebidos e novos tipos de informação possam ser extraídas através da instalação de novos conjuntos de algoritmos, agrupados em módulos.

Completando a tecnologia estão os componentes de software e padrões de comunicação que implementam o serviço de atendimento às consultas e geração de respostas.

5.2 Conceitos essenciais

Banco de dados OLAP

A tecnologia Tinycubes funciona como um banco de dados OLAP com valores armazenados em memória para evitar que o tempo de resposta a consultas possa ser influenciado pela latência inerente aos dispositivos de armazenamento de massa, como Hard-Disks, sejam eles baseados nos tradicionais discos giratórios ou mesmo nos rápidos e modernos discos SSD (*Solid State Disk*).

Tal como num banco de dados tradicional, a tecnologia Tinycubes permite que novos dados sejam recebidos, instalados e eventualmente removidos. E tal como num banco de dados OLAP, as consultas submetidas demandam respostas contendo informações resultantes da agregação dos valores armazenados, como por exemplo, histogramas, médias etc. Ressalta-se, no entanto, que os valores armazenados num banco de dados OLAP, e que produzirão respostas contendo informações de interesse, não precisam ser cópias dos dados recebidos.

Consultas restritivas

Como em qualquer banco de dados, um Banco de dados OLAP recebe consultas que, de alguma forma, podem restringir o subconjunto dos dados nos quais as respostas serão baseadas. Por exemplo, considere um banco de dados OLAP que recebe registros idênticos ao do exemplo na Figura 5.4. Enquanto todos os usuários podem estar interessados em resultados, mês a mês (o que implica em restrição dos meses a serem considerados), usuários que são gerentes municipais podem estar interessados em resultados referentes apenas às suas cidades, o que implica na restrição das cidades a serem consideradas.

Respostas agregadoras

As respostas de um banco de dados OLAP tipicamente são medidas de resumo dos dados recebidos pelo banco ou, dito de outra forma, são respostas agregadoras de valores. Ainda utilizando registros idênticos aos da Figura 5.4, exemplos típicos de consultas OLAP seriam: "Quantos produtos foram vendidos no mês X?", Qual a média de venda do produto

X no mês Y? ou mesmo "Exiba um histograma dos produtos mais vendidos no período entre o mês X e o mês Y".

5.3 Estrutura de dados

5.3.1 Índice e terminais

A estrutura de dados Tinycubes foi desenvolvida para atender aos requisitos de tempo e de tipo de resposta. Essa estrutura é organizada em duas partes. A primeira parte é um **índice** com características especiais, similar a uma árvore de busca. O percorrimento desse índice, utilizando as restrições definidas na consulta, sempre leva a segunda parte da estrutura, os elementos terminais. Cada **terminal** é responsável por armazenar valores que produzirão as informações solicitadas pelas consultas. A Figura 5.2 exibe um diagrama esquemático de um tinycube, onde o índice que leva até os terminais é representado por um triângulo com um pequeno círculo no topo, que representa um objeto especial, onde o índice se origina, denominado **Maxiroot**.

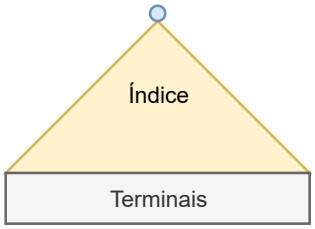
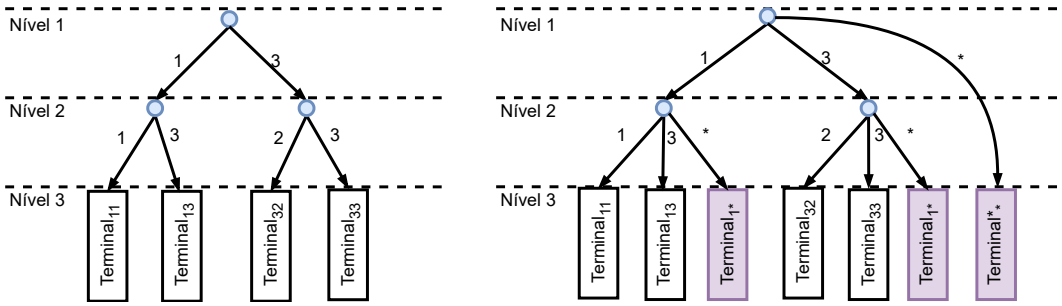


Figura 5.2: Índice e Terminais

É importante ressaltar que, embora seja aceitável, não é exato afirmar que a estrutura armazena os dados coletados. O que ocorre realmente, é que os dados, uma vez recebidos, são eventualmente transformados e depois organizados em uma estrutura de dados chamada Record. Todos os algoritmos utilizados para alteração da estrutura utilizam somente os dados presentes nesses records. Os algoritmos que realizam a construção e manutenção do índice utiliza valores obtidos à partir dos records, mas não necessariamente nem os dados originais e nem os presentes nos records. Da mesma forma, os algoritmos utilizados pelos terminais envolvidos na produção de informação armazenam valores extraídos dos records que podem ser distintos dos dados originais ou nos records.

Tinycubes vs tinycubes

Também importante fazer a distinção entre a estrutura de dados Tinycubes, um modelo conceitual baseado em regras bem definidas, e os objetos que são a manifestação



(a) Árvore de busca convencional (b) Tinycubes (simplificado)

Figura 5.3: Diferenças em relação a uma árvore de busca

computacional dessas regras, os **tinycubes**. Cada objeto, uma instância da estrutura, é individualmente chamado de **tinycube**. São esses tinycubes que efetivamente armazenarão os valores e retornarão as informações desejadas.

Localizador, Endereço e Pré-agregação

A Figura 5.3 ilustra uma das principais diferenças entre uma árvore de busca convencional e a estrutura Tinycubes. Na Figura 5.3(a), vê-se uma árvore tradicional, com os valores utilizados na busca armazenados nas arestas. A Figura 5.3(b), exibe um tinycube com os mesmos elementos da árvore convencional na figura anterior, porém contendo itens adicionais. Antes de detalhar o propósito destes elementos, é importante definir o conceito de **endereço**.

Observando a Figura 5.3 (b) vemos que os elementos adicionais em relação a (a). As arestas com valores simbólicos "*", representam o mesmo conceito de agregação apresentado nas seções 2.1 e 4.1. Ao se percorrer uma dessas arestas, atinge-se um vértice que armazena uma agregação dos valores de outros vértices. A vantagem desta abordagem com relação ao tempo de acesso decorre do fato de ser possível, durante uma consulta, acessar apenas uma informação de totalização de outros elementos, ao invés de precisar acessar cada um deles.

Em ambas as ilustrações da Figura 5.3, para se atingir o elemento terminal identificado como Terminal_{13}, à partir do vértice inicial, usa-se a aresta com valor 1 para atingir o próximo vértice para, em seguida, usar a valor 3 para chegar ao elemento. A sequência de valores de arestas que deve ser utilizada para atingir um elemento terminal é chamada de **localizador**. Quando a sequência não contém valores simbólicos, é chamada de "endereço".

Records

Os dados produzidos no mundo exterior têm de ser "organizados" antes de serem efetivamente utilizados pela estrutura de dados.

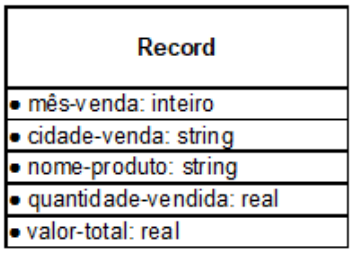


Figura 5.4: Exemplo de Record

Do ponto de vista dos algoritmos associados à estrutura, os dados sempre estão dispostos como **records** ("registros" em português), uma estrutura de dados contendo campos nomeados que armazenam valores com tipos possivelmente distintos entre si. Cada vez que se projeta uma solução usando Tinycubes, é necessário definir o conteúdo do record que conterá os dados que estarão disponíveis para utilização pelos algoritmos associados à estrutura.

Contents e Containers

Os valores armazenados em cada terminal são organizados em Contents. Cada Content é composto por um conjunto não vazio e eventualmente unitário de valores que são utilizados para gerar uma única informação. Por exemplo, um determinado Content pode conter um inteiro correspondente ao número de vezes que alguma situação foi identificada, ou, em outro exemplo, pode acumular a soma total de uma determinada medida, como valor da venda de um produto.

Existe um tipo especial de Content denominado "Container", que é capaz de armazenar outros Contents dentro de "si" mesmo. Cada Content armazenado dentro de um Container, recebe um número idenficador que é utilizado para localizá-lo posteriormente. Containers existem para oferecer um nível adicional de organização da informação. Na prática, Containers costumam ser utilizados para armazenar informações envolvendo sequências de valores, como séries temporais.

Extração de Informação

A extração de informação é realizada através da aplicação de uma função de extração sobre um Content. Para cada tipo de informação desejada existe um Content e uma

função de extração de informação específicos. Por exemplo, existem funções que extraem valores de contagem, médias, variâncias, somatórios.

5.4 Conclusão

Esta seção apresentou uma visão superficial da estrutura e da tecnologia associada ao Tinycubes. Dada a amplitude e complexidade de toda tecnologia, a sua especificação completa será realizada em vários capítulos. O capitulo 6 descreve a estrutura Tinycubes, incluindo uma abordagem formal e precisa das operações de manipulação de dados pela estrutura. O Capítulo 7 apresenta os mecanismos de modularização da tecnologia que possibilitam a introdução de novos tipos de dados, a utilização de métodos alternativos de produção de informação e a própria geração de novos tipos de informação. O Capítulo 8 apresenta os demais componente que são necessários ao funcionamento da tecnologia.

Capítulo 6

Tinycubes - Estrutura de Dados

Este capítulo apresenta a especificação formal da estrutura de dados Tinycubes utilizando conceitos da Teoria dos Grafos (ver Seção 2.4) ajustados para utilização em estruturas de dados (ver Seção 2.5). Neste capítulo e doravante, os termos **Grafo** e **Árvore**, referem-se às estruturas de dados e os termos grafo e árvore, referem-se a instâncias das estruturas Grafo e Árvore.

6.1 Schema e Grafo dimensional

6.1.1 Schema

A estrutura Tinycubes é complexa e adaptável a diversos cenários utilizando vários parâmetros para controlam a sua construção. Por uma questão de conveniência, é interessante agrupar todos esses parâmetros numa estrutura conceitual unificadora denominada **Schema**. A equação 6.1 define um Schema como um objeto matemático que relaciona os parâmetros e elementos utilizados na construção de instâncias da estrutura Tinycubes.

$$Schema =< \mathcal{R}, L, \mathcal{F}, \mathcal{S}, D, \mathcal{H} > \tag{6.1}$$

$\boldsymbol{\mathcal{R}}$ Record, estrutura de dados que dispõe os dados recebidos em forma apropriada

$\boldsymbol{D}$ Número inteiro positivo que determina a **Dimensão do Tinycubes**

$\boldsymbol{\mathcal{H}}$ *Array* com D elementos contendo as alturas dimensionais do grafo [ver **??**]

$\boldsymbol{L}$ Número inteiro positivo que indica o **Comprimento do Tinycubes**

$\boldsymbol{\mathcal{F}}$ *Array* com L funções mapeadoras [ver 6.2.2]

$\boldsymbol{\mathcal{S}}$ Array não vazio cujo os elementos são SContents [ver 6.2.5]

Definição 6.1.1. Dimensão de um grafo dimensional. O valor da constante D determina tanto o tamanho do *array* $\mathfrak{D}$ quanto o número de dimensões do grafo dimensional que utiliza o *array*.

Definição 6.1.2. Dimensão de um Tinycubes é dada pela dimensão do grafo dimensional. Na prática, utiliza-se a nomenclatura, "Tinycubes com dimensão d" ou "Tinycubes com $D = d$" onde $d \in \mathbb{N}^+$.

6.1.2 Grafo dimensional - G

Os objetos criados por uma estrutura de dados Tinycubes podem ser representados (modelados) por grafos contendo algumas extensões que serão descritas a seguir.

Definição 6.1.3. Grafo dimensional é um Grafo dirigido fracamente conexo com características adicionais que possibilitam utilizá-lo na recuperação acelerada de informações à partir de dados recebidos previamente.

$$G_d =< V, E, v_t, Schema > \tag{6.2}$$

A equação 6.2 define a estrutura matemática que enumera os elementos que compõem o grafo dimensional G_d que modela um Tinycubes. Os elementos são:

Schema estrutura com parâmetros utilizados na construção do grafo (ver 6.1.1)

$\boldsymbol{V}$ é um conjunto de vértices dimensionais

$\boldsymbol{E}$ é um conjunto de arestas (*Edges*) direcionais dimensionais

$\boldsymbol{v_{tr}}$ identifica um vértice especial de V, denominado **Maxiroot**

Além da característica dimensional e diferentemente dos grafos tradicionais, os vértices e as arestas de um Tinycubes possuem características adicionais que facilitam a recuperação acelerada de informações.

Esta seção descreve os elementos constituintes do grafo dimensional que modela um Tinycubes. (Introdução ao conceito de dimensão, vértices dimensionais, arestas dimensionais, dimensões)

6.2 Elementos do Schema

6.2.1 Record

Os dados relevantes para extração de informações de interesse são gerados no mundo real de diversos modos e codificados de diversas formas. Para que esses dados possam ser utilizados em um Tinycubes na recuperação de informações é necessário organizá-los de alguma forma padronizada. Com essa motivação, define-se o conceito de **Record**, uma estrutura de dados auxiliar que dispõe os dados de entrada de uma forma apropriada para utilização pelos algoritmos de um Tinycubes.

O Exemplo a seguir, exibe um *Record* e dois records (objetos) definidos em conformidade com esse *Record*:

```
Record = (idade : NúmeroInteiro, salário: NúmeroReal )
r1 = (idade= 35, salário= 3421.16)
r2 = (idade= 42, salário= 5896.32)
```

6.2.2 Funções mapeadoras

Esta seção apresenta o mecanismo utilizado para obtenção dos valores que serão armazenados em arestas CHILD à partir do conteúdo dos records de entrada.

Definição 6.2.1. Uma **Função mapeadora** é uma função que, ao receber um record com dados, retorna um número inteiro. Matematicamente, uma função mapeadora é descrita como:

$$map : Record \mapsto \mathbb{Z} \tag{6.3}$$

As funções mapeadora são responsáveis pela conversão dos dados de entrada em valores que serão armazenados nas arestas. O exemplo a seguir ilustra como uma função mapeadora pode ser implementada utilizando a composição do *Record* definido no exemplo anterior.

A função mapeadora $MapIdade$ recebe um record no formato definido pela estrutura *Record* e retorna o valor inteiro: 1, se o valor no campo idade do record é inferior a 18 ou superior a 65; 2, se o valor do campo idade é inferior a 35 e; 3 se o valor é superior está entre 35 e 65 inclusive; 4, se o valor é superior a 65. Essa função mapeia uma idade em

um dentre quatro possíveis valores que representam **categorias** lógicas de classificação, e por essa razão os valores armazenados nas arestas são chamados de **valores categóricos**.

Definição 6.2.2. Valor categórico, no contexto de Tinycubes, é um valor inteiro que representa uma categoria lógica de valores de um ou mais dados de entrada. Por exemplo, o valor de um salário podem ser classificado (categorizado) como baixo, médio ou alto, que por sua vez podem corresponder aos valores inteiros 1, 2 e 3, respectivamente.

Array de mapeamento

Como existem diversas arestas CHILD, dispostas em diversos níveis dimensionais, são necessárias diversas funções mapeadoras.

A definição 6.2.1 descreve formalmente como o valor que tupla é mapeada numa altura

Definição 6.2.3. Em um Tinycubes com dimensão D, o valor armazenado em uma aresta CHILD a na dimensão d ($d \in [1, D]$) e com altura h_a ($h_a \in [1, H_d]$) calculada como $h_a = H(a)$ é computado aplicando-se a função mapeadora $map_{d,h} : T \mapsto \mathbb{Z}$ sobre um record $r \in Record$.

6.2.3 Contents

Esta seção apresenta especificações formais para as operações que manipulam os valores presentes nos vértices terminais, doravante chamados de Contents. Content e "Decomposable Aggregate Functions" [77, 32] compartilham o mesmo conceito, porém Content é uma abstração que isola a estrutura de dados (e outras partes da tecnologia) de detalhes de implementação, exigindo apenas que um conjunto mínimo de propriedades esteja corretamente implementado para o correto funcionamento da tecnologia. Utilizando este nível de abstração, é possível definir conceitos, propriedades e operações de forma abstrata e independente do caso concreto. A partir dessas definições, torna-se possível criar um sistema de modularização capaz de permitir a instalação de novas implementações de Contents já existentes (para atualizar técnicas) ou mesmo, a implementação de Contents que definem novos tipos de informação.

Ressalta-se que, embora vértices terminais possam armazenar diferentes Contents simultaneamente, para facilitar a compreensão e sem perda de generalidade, as especificações das operações desta seção é apresentada como se houvesse apenas um subcube por terminal e apenas um conjunto de dados relevantes a ser extraído dos records.

6.2.3.1 Content

Definição 6.2.4. Content. Define-se como **Content**, uma estrutura de dados, similar a um Record, que armazena valores utilizados para produção uma informação.

6.2.3.2 Inserção em um content

As seguintes funções são utilizadas para realizar a inserção de dados relevantes presentes em um record no content de um terminal.

$$select : Record \mapsto Suite \tag{6.4}$$

$$apply : Suite \times Content \mapsto Content \tag{6.5}$$

onde:

Record *Record* conforme definido na Seção 6.2.1

Suite conjunto de dados relevantes extraídos de um *Record*

Content conjunto de valores utilizados para gerar informação (definição 6.3.7).

select função que extrai dados de um *record* retornando um *Suite*

apply função que gera um novo *content* à partir de um *content* existente

O processo de inserção de dados em um terminal consiste na extração (com eventual transformação) dos dados relevantes de um *record* utilizando uma função *select*, produzindo como resultado um *suite* contendo os valores a serem utilizados. Em seguida, utilizando a função *apply*, o processo gera um novo *content* à partir do *content* existente e do *suite* recém gerado. Esse processo pode ser representado matematicamente através das seguintes equações:

$$suite \leftarrow select(record) \qquad record \in Record \tag{6.6}$$

$$content \leftarrow apply(suite, content) \qquad content \in Content \tag{6.7}$$

Após inserções de k records em um terminal, o valor final do content (C_k) do terminal pode ser computado através da sucessiva aplicação das funções *select* e *apply*.

Sejam:

$\boldsymbol{C_0}$ um elemento distinto em *Content* denominado "content inicial"

$\boldsymbol{R_k}$ uma sequência de k records $(r_1, r_2, ..., r_k)$ com $r_i \in Record, i \in [1, k]$

$\boldsymbol{C_k}$ o content associado ao k-ésimo record de uma sequencia de records

A fórmula para o cálculo do content C_k é definida como:

$$C_k = \begin{cases} C_0 & k = 0 \\ apply(select(r_k), C_{k-1}) & \text{caso contrário} \end{cases} \tag{6.8}$$

Note que a operação de "inserção" não "insere" dados no content, e sim provoca uma transformação no valor do content por conta dos dados utilizados.

6.2.3.3 Extração de informação de um content

A informação retornada em uma consulta é produzida à partir dos valores armazenados em contents. O processo de transformação desses valores armazenados em uma informação que atenda a consulta realizada é modelado matematicamente pela função *extract*.

$$extract : Content \mapsto Info \tag{6.9}$$

onde:

$\boldsymbol{Info}$ tipo do valor a ser retornado em uma consulta

$\boldsymbol{extract}$ função que transforma um content em um valor a ser retornado

Em muitos casos, os valores armazenados em um content já correspondem à informação desejada, como no caso onde o content armazena a contagem de ocorrências de um evento. Nesses casos, a função *extract* apenas reproduz o valor armazenado no content. Em outros casos, o content armazena um conjunto de valores que deve ser computado antes de se transformar em informação. Por exemplo, quando a informação desejada é a média de alguma medida, o content deve armazenar a soma acumulada das medidas *sum* e a contagem de medições *counter*. Neste caso, a função *extract* retornará a divisão de *sum* por *counter*.

6.2.3.4 Relevância da ordem de inserção de records

A ordem de inserção dos records na estrutura pode influenciar ou não qual informação será retornada pelas consultas. O fato da função *apply* ser ou não comutativa determina se a ordem de inserção dos records pode afetar a geração das informações resultantes.

A função *apply* é comutativa se, para todo $r_1, r_2 \in Record$ e todo $c \in Content$, é possível afirmar que:

$$apply(s_2, apply(s_1, c)) = apply(s_1, apply(s_2, c)) \tag{6.10}$$

onde:

$$s_1 \leftarrow select(r_1), s_2 \leftarrow select(r_2)$$

Se a função *apply* é comutativa, é fácil perceber que duas sequências contendo os mesmos records k, porém dispostos numa ordem diferente, produzirão o mesmo valor de content porque sempre será possível alterar uma das sequências usando comutação, como no algoritmo BubbleSort, até que ela fique exatamente igual a outra e, portanto, gerando o mesmo valor final de content (ver fórmula 6.8). Assim, se a função *apply* é comutativa, pode-se afirmar que a ordem em que os records se apresentam para inserção não afeta a informação que será retornada pelas consultas posteriormente.

6.2.3.5 Remoção de dados de um content

Formalmente, a remoção de dados de um content pode ser especificada pela função *unapply*, que pode ser entendida como uma função que reverte os efeitos oriundos da aplicação da função apply. De fato, a operação de remoção não "remove" os dados de um content porque esses dados provavelmente não estão armazenados no content. O que a operação de **remoção** faz é desfazer o efeito da propriamente dos dados no content.

$$unapply : Suite \times Content \mapsto Content \tag{6.11}$$

$$suite \leftarrow select(record) \tag{6.12}$$

$$content \leftarrow unapply(suite, apply(suite, content)) \tag{6.13}$$

A função *unapply* só é válida se a equação 6.13 for válida para todo $record \in Record$ e todo $content \in Content$.

É importante observar que, se a função *apply* não for comutativa, não é possível garantir que os dados de um record, uma vez inseridos (aplicados) em um content, possam ser removidos após a inserção dos dados de outros records. Formalmente, sejam $s_1 \leftarrow$

$select(r_1)$ e $s_2 \leftarrow select(r_2)$, onde $r_1, r_2 \in Record$. Não é possível garantir que:

$$content \leftarrow unapply(s_2, unapply(s_1, apply(s_2, apply(s_1, content)))) \tag{6.14}$$

6.2.3.6 Agregação de dois Contents - Função *merge*

Em um banco de dados OLAP, é natural que a realização de uma consulta tenha de fundir informações oriundas de terminais diferentes, por exemplo, quando deseja-se o total combinado de dois elementos. Além disso, a estrutura Tinycubes é definida com a expectativa de que vértices terminais atingíveis através de arestas agregadoras possuam contents que correspondam a "combinação" dos contents dos vértices terminais atingíveis através das arestas seletoras irmãs (definição 2.4.16) dessas arestas agregadoras.

Isso nem sempre é verdade, pois depende tando da forma como as funções de manipulação de contents foram construídas quanto da forma como a "agregação" de contents é realizada.

Esta seção descreve formalmente em que condições a agregação de contents em terminais atingíveis por arestas seletoras produzirá o mesmo content armazenado em arestas agregadoras, garantindo que a informação será consistente, independentemente do(s) contents utilizado.

Define-se a função *merge*, responsável por produzir um novo content à partir da agregação de dois outros contents, como:

$$merge : Content \times Content \mapsto Content \tag{6.15}$$

Sendo que essa função *merge* será considerada comutativa se:

$$merge(a, b) = merge(b, a) \qquad a, b \in Content \tag{6.16}$$

Como requisitos iniciais necessários para que a função *merge* produza resultados consistentes, exige-se que:

1. a função *merge* seja comutativa;
2. a função *apply*, com o mesmo conjunto *Content* da função *merge*, seja comutativa.

A especificação do requisito final para garantir que a função *merge* realize uma agregação consistente de contents exige a definição alguns elementos adicionais.

Inicialmente, considere:

$\boldsymbol{R_n}$ sequência de records $(r_1, r_2, MID..r_n)$ onde $r_i \in Record, i \in [1, n]$

$\boldsymbol{SeqR_n}$ conjunto de todas as sequências contendo n records $(R_n \in SeqR_n)$

$\boldsymbol{C_k}$ content computado como na equação 6.8

Define-se a função auxiliar $flip$ que aleatoriamente retorna os valores 1 ou 2.

$$flip: \quad \mapsto \{1, 2\} \tag{6.17}$$

Definem-se os contents C_k^1 e C_k^2, com $(k \in [1, n])$, calculados conforme a fórmula abaixo.

$$C_k^i = \begin{cases} C_0 & k = 0 \\ apply(select(r_k), C_{k-1}^i) & flip() = i \\ C_{k-1}^i & \text{caso contrário} \end{cases}$$

Com base nessas definições, para que a função $merge$ possa realizar agregações consistentes, exige-se também:

$$merge(C_k^1, C_k^2) = C_k \qquad \forall C_k^1, C_k^2 \in Content \tag{6.18}$$

6.2.3.7 Agregação de múltiplos contents

Para estender o requisito definido pela expressão 6.18 de forma a poder aplicá-lo em situações que exigem a agregação de vários contents, são necessárias definições de dois novos operadores.

Define-se o operador $\oplus$ como uma versão da função $merge$ na forma de "operador binário".

$$\oplus : Content \times Content \mapsto Content \qquad \text{tal que } a \oplus b = merge(a, b) \tag{6.19}$$

Define-se também o operador $\mathcal{M}_i^n$ como uma versão da função $merge$ aplicável a uma sequência de n contents da mesma forma que o operador somatório $\sum_i^n$ é aplicável a uma

sequência de n números.

$$\mathcal{M}_i^n c_i = \begin{cases} c_1 & n = 1 \\ c_1 \oplus c_2 ... \oplus c_n & n > 1 \end{cases} \tag{6.20}$$

Com base na definição de R_n realizada na Seção 6.2.3.6, considere:

SetR o conjunto de todos os elementos em R_n (descartando repetições)

PartR uma partição de $SetR$ definida como $PartR = \{P^i\}$ onde $i \in [1, m]$

Define-se a função *partition*, associada a partição $PartR$, que, ao receber uma sequência R contendo n records ($R_n \in SeqR_n$) e um inteiro k ($k < n$), retorna um valor inteiro i que identifica uma das m partições P^i em $PartR$ ($i \in [1, m]$)[1].

$$partition : SeqR_n \times Inteiro \mapsto Inteiro \tag{6.21}$$

Seja o content C_k^i, definido como o content associado à partição P^i e ao k-ésimo record da sequencia R, calculado utilizando a fórmula:

$$C_k^i = \begin{cases} C_0 & k = 0 \\ C_{k-1}^i & partition(R, k) \neq i \\ apply(select(r_k), C_{k-1}^i) & \text{caso contrário} \end{cases}$$

Exige-se que a agregação de todos contents associados a cada $P^i \in PartR$, calculados à partir de uma sequência de records R_n, seja igual ao content calculado diretamente à partir de R, o que pode ser formalmente expresso como:

$$\mathcal{M}_i^m C_k^i = C_k \qquad \forall k \in [1, n], \forall PartR \tag{6.22}$$

6.2.3.8 Exemplos de Contents

Esta seção apresenta exemplos de aplicação das funções de manipulação de contents. Todos os exemplos listados aqui baseiam-se em um cenário onde os dados são gerados por um sistema que monitora uma redes de computadores coletando diversas informações operacionais. Neste cenário, foi decidido que apenas os dados referentes ao número de

[1] Um índice tipo árvore pode ser visto como uma materialização de uma partição do conjunto de elementos armazenáveis na árvore, onde cada folha está associada a um dos subconjuntos que formam a partição

bytes recebidos em cada pacotes de dados são relevantes e que as seguintes informações deveriam estar disponíveis nas respostas às consultas:

- Quantidade de pacotes recebidos
- Total acumulado de bytes recebidos
- Tamanho médio do pacote de dados
- Maior quantidade de bytes em um pacote

A geração de cada uma dessas informações utiliza um *Content* próprio que, para ser manipulado, necessita dos diversos elementos, conjuntos e funções, definidos ao longo desta seção. As tabelas 6.1 e 6.2 contêm exemplos que ilustram como os elementos necessários podem ser construídos para gerar cada uma das informações desejadas.

Quantidade de pacotes recebidos	
Conjuntos e Elementos	
Record	(bytesRecebidos)
Suite	{dummy} // não utilizado
Content	{counter}
Info	{counter}
C_0	{0}
Funções	
select(r)	↦ { 0 } // não utilizado
apply(s, c)	↦ { c.counter + 1 }
unapply(s, c)	↦ { c.counter - 1}
extract(c)	↦ { c.counter }
merge(a, b)	↦ { a.counter + b.counter }

Total de bytes recebidos	
Conjuntos e Elementos	
Record	(bytesRecebidos)
Suite	{value}
Content	{sum}
Info	{totalAcumulado}
C_0	{0}
Funções	
select(r)	↦ {bytesRecebidos}
apply(s, c)	↦ { c.sum + s.value }
unapply(s, c)	↦ { c.sum - s.value }
extract(c)	↦ { c.sum }
merge(a, b)	↦ { a.sum + b.sum }

Tabela 6.1: Total de pacotes e Total de bytes recebidos

Tamanho médio do pacote (bytes/pacote)	
Conjuntos e Elementos	
Record	(bytesRecebidos)
Suite	{value}
Content	{counter, sum}
Info	{media}
C_0	{0, 0}
Funções	
select(r)	↦ {bytesRecebidos}
apply(s, c)	↦ { c.counter+1, c.sum + s.value}
unapply(s, c)	↦ { c.counter-1, c.sum - s.value}
extract(c)	↦ { c.sum / c.counter }
merge(a, b)	↦ { a.counter + b.counter, a.sum + b.sum}

Maior quantidade de bytes em um pacote	
Conjuntos e Elementos	
Record	(bytesRecebidos)
Suite	{value}
Content	{max}
Info	{max}
C_0	{-1}
Funções	
select(r)	↦ {bytes}
apply(s, c)	↦ { (if (s.value > c.max) then s.value else c.max) }
unapply(s, c)	↦ **impossível calcular corretamente sempre**
extract(c)	↦ { c.max }
merge(a, b)	↦ { (if (a.max > b.max) then a.max else b.max) }

Tabela 6.2: Tamanho médio do pacote e Maior quantidade de bytes

6.2.4 Containers

Um Container é um tipo especial de Content que oferece, além das funções já oferecidas por um content, as seguintes funções.

$$getIndex : Suite \mapsto Index \tag{6.23}$$

$$getContent : Index \mapsto Content \tag{6.24}$$

$$setContent : Index \times Content \mapsto Content \tag{6.25}$$

onde:

Suite conjunto de dados relevantes extraídos de um *Record*

Index valor inteiro

Content conjunto de valores utilizados para gerar informação (definição 6.3.7).

Exigem-se as seguintes propriedades dessas funções:

$$setContent(i, C) = C \tag{6.26}$$

$$getContent(i, setContent(i, C)) = C \qquad \forall i \in \mathbb{Z}, C \in Content \tag{6.27}$$

O objetivo de um Container é definir uma dimensão adicional, fora da estrutura do Grafo que implementa o índice do Tinycubes, respeitando o padrão estabelecido pelos Contents. Para atingir a esse objetivo, os Containters são definidos como extensões dos Contents. Deste modo, as funções específicas para Containers são utilizadas internamente na implementação das funções *apply* e *unapply* para Containers. Por exemplo:

$$\begin{aligned}
&containerApply : Suite \times Content \mapsto Content \\
&\quad Index = \mathbf{getIndex}(Suite) \\
&\quad \rightarrow \mathbf{setContent}(Index, apply(Suite, \mathbf{getContent}(Index)))
\end{aligned}$$

$$\begin{aligned}
&containerUnapply : Suite \times Content \mapsto Content \\
&\quad Index = \mathbf{getIndex}(Suite) \\
&\quad \rightarrow \mathbf{setContent}(Index, unapply(Suite, \mathbf{getContent}(Index)))
\end{aligned}$$

6.2.5 SContent

Define-se como SContent a estrutura que agrupa todos os elementos necessários para utilização de um Content por um Tinycubes. SContents são importantes para especificação dos vértices terminais numa estrutura Tinycubes.

$$SContent =< Content, select, insert, remove, extract, merge > \tag{6.28}$$

Content tal como definido na Seção 6.2.3.1;

select tal como definido na Seção 6.2.3.2;

insert corresponde à função *apply*, tal como definido na Seção 6.2.3.2;

remove corresponde à função *unapply*, tal como definido na Seção 6.2.3.5;

extract tal como definido na Seção 6.2.3.3;

merge tal como definido na Seção 6.2.3.6;

6.3 Elementos do Grafo Dimensional

6.3.1 Definições e propriedades iniciais

Definição 6.3.1. Maxiroot. Define-se como Maxiroot, o único vértice de um tinycube que não é destino de nenhuma aresta. Além disso, esse vértice é permanente e invariável ao longo do tempo e também, como vários outros vértices, não armazena nenhum valor.

Definição 6.3.2. tinycube vazio. Define-se como "tinycube vazio"[2], o tinycube que contém apenas o Maxiroot e nenhuma aresta. Nessa situação, pode-se dizer que "o tinycube está vazio".

Definição 6.3.3. tinycube regular. Define-se como "tinycube regular" um tinycube que não esteja vazio.

Definição 6.3.4. dimensão. Para um tinycube, uma dimensão d é apenas um número inteiro positivo menor ou igual a $\mathbf{D} + 1$ (quantidade de elementos do *array* $\mathfrak{H}$).

As dimensões são utilizadas na definição de propriedades que ajudam a impor uma organização restritiva para o grafo dimensional. A seguir são listadas as propriedades fundamentais envolvendo dimensões referentes ao grafo que modela um Tinycubes.

[2]Não confundir com "grafo vazio" da Teoria dos Grafos, onde não existe nenhum vértice. O termo "vazio" utilizado aqui, corresponderia a "grafo trivial" na Teoria dos Grafos

Propriedade 6.3.1. dimensionalidade. Cada um dos vértices e arestas do grafo G sempre está vinculado a uma dimensão d, sendo que $d \in [1, D+1]$. Quando um elemento (vértice ou aresta) possui um vínculo com uma dimensão d, diz-se que "o elemento está na dimensão d" ou que "o elemento pertence à dimensão d"

Definição 6.3.5. notação de dimensionalidade. Um elemento x na dimensão d pode ser representado como $x(d)$.

6.3.2 Vértices dimensionais

Um vértice dimensional possui características adicionais relacionadas às dimensões do Tinycubes. Um vértice dimensional pode ser de um dentre dois tipos: **vértice regular** ou **vértice terminal**.

Definição 6.3.6. Vértice Regular. Define-se como **vértice regular**, um vértice que não armazena nenhum valor. O Maxiroot é considerado um vértice regular por não armazenar valores. Todos os vértices regulares, à exceção do Maxiroot, tem de ser origem e destino de uma ou mais arestas. Um Maxiroot não vazio é apenas origem de arestas, nunca o destino. Dimensionalmente, vértices regulares sempre estão numa dimensão d, onde $d \in [1, D]$.

Definição 6.3.7. Vértice Terminal. Define-se como **vértice terminal**, um vértice que armazena valores que são utilizados para produção de informações. Vértices terminais sempre são destino de alguma aresta mas não são origem de nenhuma, ou seja, terminam todos os caminhos que o alcançam[3]. De fato, todos os caminhos em um Tinycubes podem acabar em um vértice terminal. Dimensionalmente, vértices terminais sempre estão na dimensão $D+1$

Definição 6.3.8. Define-se como **vértice TOP**, um vértice numa dimensão d que não é filho de um vértice na mesma dimensão d. Quando um vértice v é um vértice TOP, pode se dizer que "v é TOP".

Note que, de acordo com essa definição, o Maxiroot é TOP porque não é apontado por nenhuma aresta, logo não é filho de um vértice com a mesma dimensão que a sua. Ao mesmo tempo, todos os vértices terminais tem de ser TOP porque estão na dimensão $D+1$ e nenhuma aresta está em $D+1$ por terem origem em vértices regulares que, no máximo, estão na dimensão D.

[3]Para fins de comparação, sob o ponto de vista da análise de "Redes de Fluxo"(*Flow Networks*), o vértice Maxiroot seria a única fonte (*source*) no grafo enquanto os vértices terminais seriam classificados como sumidouros (*sinks*).

Definição 6.3.9. Define-se como **vértice MID**, um vértice numa dimensão d que seja filho de um vértice na mesma dimensão d. Quando um vértice v é um vértice MID, pode se dizer que "v é MID".

Observe que todos os vértices MID têm de ser regulares porque, se fossem terminais, que sempre estão na dimensão D+1, teriam arestas na dimensão D+1 apontando para eles, o que é impossível.

Figura 6.1: Representação gráfica para vértices de um Tinycubes

A Figura 6.1 exibe mais à esquerda um vértice (regular ou terminal) que é representado graficamente por um círculo, que sinaliza que é um vértice regular. A seguir, vê-se um vértice (círculo) com uma identificação interna que serve apenas para facilitar explicações, não tendo nenhum significado para a estrutura. Ao centro-direita, a figura exibe uma forma alternativa de ilustração de um vértice terminal representado por um retângulo de cantos arredondados. Na notação utilizada neste trabalho, a informação armazenada em um vértice (terminal) é simbolizada pelo pequeno retângulo branco no interior do vértice. A notação também utiliza, para facilitar a compreensão, identificadores no padrão T_i em vértices terminais que, apesar de exibidos nas representações, não estão presentes na estrutura. Opcionalmente, vértices regulares podem ser identificados para melhor compreensão, como pode ser visto no círculo mais à direita da Figura 2.3.

6.3.3 Arestas dimensionais

Uma aresta dimensional possui características adicionais relacionadas às dimensões do Tinycubes.

Propriedade 6.3.2. relação vértice → aresta. Toda a aresta com o vértice origem na dimensão d também está na dimensão d.

Propriedade 6.3.3. relação aresta → vértice. Uma aresta na dimensão d sempre aponta para um vértice na dimensão d ou na dimensão $(d+1)$. Note que o vértice destino

de uma aresta será considerado TOP estiver na dimensão seguinte a do vértice origem e MID caso contrário.

Propriedade 6.3.4. relação aresta → vértice. Os vértices destino de todas as arestas com um mesmo vértice origem têm de estar na mesma dimensão.

As arestas dimensionais possuem dois atributos independentes que são fundamentais para os algoritmos que manipulam a estrutura. O primeiro atributo sinaliza para os algoritmos se a aresta é "logicamente responsável" pela existência do seu vértice destino ou não, possuindo os valores PROPER e SHARED.

Definição 6.3.10. aresta **PROPER** Define-se como aresta PROPER (ou aresta tipo PROPER), uma aresta que é "vinculada" à existência do seu vértice para o qual aponta por ele ter sido criado especialmente para ser seu destino. Nessas condições, pode se dizer que o "vértice origem é **dono** do vértice destino". À exceção do Maxiroot, todo vértice sempre é apontado por uma, e apenas uma, aresta PROPER. No contexto de estruturas de dados, este tipo de aresta pode ser considerado o tipo normal de aresta.

Definição 6.3.11. aresta **SHARED** Define-se como aresta SHARED (ou aresta tipo SHARED), uma aresta que tem como destino um vértice que é de responsabilidade de outra aresta, sendo, portanto, também apontado por uma aresta PROPER. À exceção do Maxiroot, todo vértice pode ser apontado por uma aresta SHARED.

O segundo atributo refere-se à finalidade da aresta durante a recuperação da informação, possuindo os valores CHILD e CUBE.

Definição 6.3.12. Define-se como aresta **CHILD** (ou aresta tipo CHILD), uma aresta que armazena um valor inteiro, denominado **OPTION**. Com base nesse valor, os algoritmos de busca decidem qual dos vértices irmãos devem escolher para continuar o caminho até atingir um vértice terminal e obter a informação a ser retornada para a consulta. Exige-se que nenhuma aresta CHILD tenha o mesmo OPTION que suas arestas irmãs. Por fazerem parte da seleção do vértice a ser escolhido para o caminho, essas arestas também são chamadas de **arestas seletoras**. Todos os vértices regulares, à exceção do Maxiroot vazio, são origem de pelo menos uma aresta CHILD.

Definição 6.3.13. Define-se como aresta **CUBE** (ou aresta tipo CUBE), uma aresta que não armazena valores e tem como destino um vértice que está sempre na dimensão seguinte a sua própria dimensão[4] (e consequentemente na dimensão seguinte a do seu vértice

[4]Dada a organização hierárquica de um Tinycubes, a cada nova dimensão, o tamanho do cubo representado diminui, daí a origem do nome CUBE

origem). Todos os vértices regulares, à exceção do Maxiroot vazio, sempre são origem de uma, e de apenas uma, aresta CUBE. Para melhor compreensão, pode-se considerar que arestas CUBE armazenam um valor simbólico representado como asterisco ('*'), mas que, de fato, não é armazenado. As arestas CUBE foram projetadas para reduzir o tempo de resposta em consultas com critérios de seleção menos restritivos do tipo "recupere o total para todos os elementos de um conjunto".

Definição 6.3.14. Define-se como aresta **I-CHILD**, uma aresta do tipo PROPER CHILD, sendo que a letra "I" que prefixa o termo "CHILD" indica que é uma aresta que referencia um vértice interno[4].

Definição 6.3.15. Define-se como aresta **I-CUBE**, uma aresta do tipo SHARED CUBE, sendo que a letra "I" que prefixa o termo "CUBE" indica que é uma aresta que referencia um datacube interno[4].

Definição 6.3.16. Define-se como aresta **X-CHILD**, uma aresta do tipo SHARED CHILD, sendo que a letra "X" que prefixa o termo "CHILD" indica que é uma aresta que referencia um datacube externo[4].

Definição 6.3.17. Define-se como aresta **X-CUBE**, uma aresta do tipo PROPER CUBE, sendo que a letra "X" que prefixa o termo "CUBE" indica que é uma aresta que referencia um datacube externo[4].

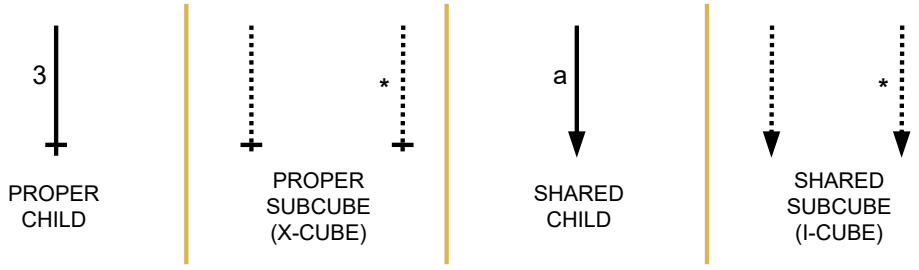


Figura 6.2: Representação gráfica para as de arestas usadas em um Tinycubes

A Figura 6.2 apresenta a notação para todos os tipos de aresta. A direção das arestas é indicada por uma marca (variação gráfica) na extremidade correspondente ao destino das linhas que representam as arestas. Nas arestas do tipo PROPER, a marca é um pequeno traço transversal a linha da aresta, enquanto as arestas do tipo SHARED possuem uma seta na extremidade. Ao mesmo tempo, arestas com linhas sólidas são arestas seletoras (tipo CHILD) enquanto arestas formadas por linhas pontilhadas são arestas agregadoras

[4] O conceito de externo e interno ficará claro quando após a definição de tinytrees, mais adiante neste capítulo

(tipo CUBE). Na notação utilizada, os valores armazenados nas arestas seletoras (OPTIONs) são exibidos próximos a elas (na figura, esses valores estão representados pelo inteiro 3 e pela constante 'a', associada a algum valor genérico). As arestas agregadoras podem exibir o caractere '*' ao seu lado ou não, uma vez que o pontilhado também indica que a aresta é dimensional.

6.3.4 Características estruturantes

Nesta seção serão conceitos e características que determinam a estrutura típica de um Grafo que modela um Tinycubes.

Definição 6.3.18. Define-se $\boldsymbol{P_d}$, **caminho completo na dimensão** $\boldsymbol{d}$, como um caminho dirigido simples iniciado em um vértice TOP $x(d)$ e terminado em um vértice TOP $y(d+1)$ passando apenas por arestas CHILD na dimensão d. Note que, por definição, um caminho completo só pode ter vértices MID entre os vértices TOP x e y.

Definição 6.3.19. O **comprimento de um caminho completo** $\boldsymbol{P_d}$, representado por $|P_d|$, é dado pela quantidade de arestas CHILD que compõem P_d.

Propriedade 6.3.5. Comprimento da dimensão d. Todos os caminhos completos possíveis em uma dimensão d têm de ter o mesmo comprimento.

Essa última propriedade determina um padrão estrutural uniforme para cada dimensão do grafo. Com base nessa propriedade, e nas definições realizadas, já é possível construir tinycubes de forma consistente.

No entanto, para especificar precisamente a forma que um Tinycubes deve assumir, é necessário definir mais alguns conceitos que possibilitarão a especificação de mais alguns parâmetros construtivos.

Alturas e afins

Definição 6.3.20. Define-se $\boldsymbol{H_d}$, **altura da dimensão d**, como o número inteiro correspondente ao comprimento de uma dimensão d. A altura da dimensão d também pode ser chamada de "altura dimensional de d".

Definição 6.3.21. Define-se **H(v)**, **altura do vértice** $\boldsymbol{v}$, como o comprimento de um caminho simples entre algum vértice TOP u e o vértice v, passando apenas por arestas CHILD e vértices MID. Note que, se o próprio v é um vértice TOP, então $u = v$ e portanto $H(v) = 0$.

Definição 6.3.22. Define-se **H(a)**, **altura da aresta** $\boldsymbol{a}$, que tem origem no vértice u, como a altura do vértice u acrescido de 1, ou seja, $H(a) = H(u) + 1$.

Definição 6.3.23. Define-se $\boldsymbol{\mathcal{H}}$, **array dimensional**, como o array com D elementos onde cada posição d ($d \in [1, D]$) tem o valor da altura da dimensão d. Matematicamente: $\mathcal{H}[d] = H_d \quad \forall d \in [1, D]$.

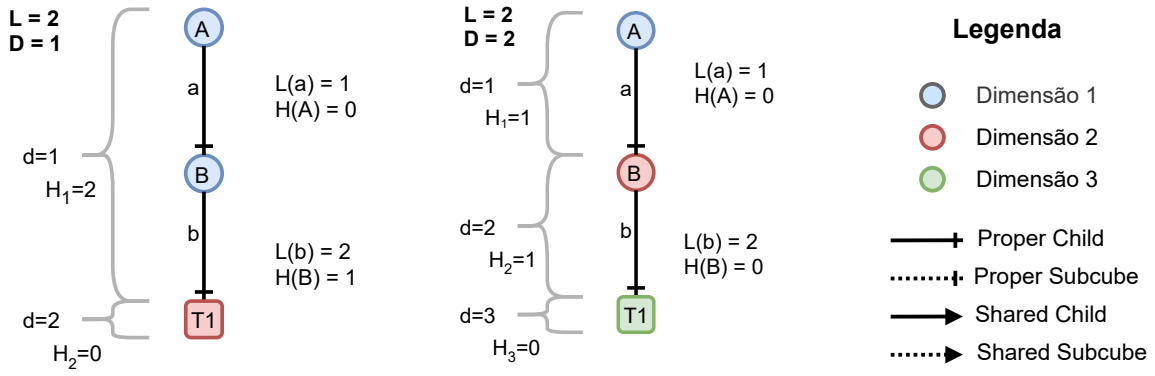
Definição 6.3.24. Define-se como **dimensão simples**, uma dimensão d cuja a altura dimensional seja 1 ($H_d = 1$), e como **dimensão complexa** ou **dimensão hierárquica**, uma dimensão d com altura dimensional maior do que 1 ($H_d > 1$).

Níveis e comprimento de um Tinycubes

Definição 6.3.25. Define-se $\boldsymbol{L_d}$, **nível da dimensão** $\boldsymbol{d}$, o valor inteiro computado como a quantidade arestas CHILD percorridas partindo-se de um Maxiroot não vazio até chegar a algum vértice TOP em d acrescido do valor 1.

Definição 6.3.26. Define-se **L(a)**, **nível da aresta** $\boldsymbol{a}$ na dimensão d como o resultado da soma do nível da dimensão d com a altura da aresta a subtraído de 1. Matematicamente: $L(a) = L_d + H(a) - 1$

Definição 6.3.27. Define-se $\boldsymbol{L}$, **comprimento de um Tinycubes**, como a soma das alturas dimensionais de cada dimensão do Tinycubes. $L = \sum_{d=1}^{D} H_d$



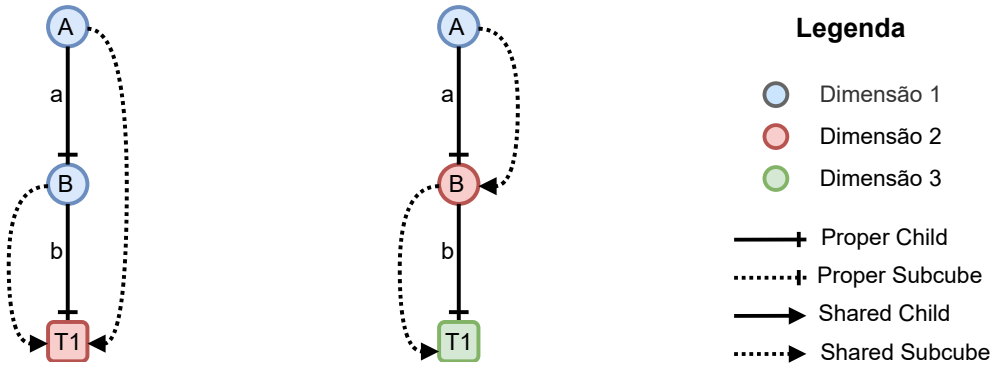
(a) Uma dimensão complexa (b) Duas dimensões simples

Figura 6.3: Fragmentos de tinycubes com L=2 mas diferentes valores de D

Endereço e Subendereço

Definição 6.3.28. Define-se como **Endereço**, um array, iniciado na posição 1, com L números inteiros positivos.

Definição 6.3.29. Define-se como **Subendereço para dimensão d**, a parte de um endereço iniciada na posição L_d do array e terminando na posição L.



(a) D=1 (b) D=2

Figura 6.4: Tinycubes completos com L=2 mas diferentes valores de D

6.3.5 Trees: Tinytrees, Maxitree, ...

Como um Tinycubes é modelado através de um Grafo, é possível construir diferentes subgrafos à partir dele. Esta seção apresenta uma forma de criação de um tipo especial de subgrafo que possibilita a definição de conceitos essenciais para construção dos algoritmos de inserção e remoção de dados na estrutura.

Definição 6.3.30. Define-se como vértice **raiz** ou **root**, um vértice regular que não é apontado por arestas CHILD. Note que esta definição implica que o Maxiroot é um vértice raiz.

Definição 6.3.31. Define-se como **subraiz** ou **subroot**, um vértice raiz que não é o Maxiroot.

Definição 6.3.32. Define-se como **subvértice**, um vértice descendente de uma subraiz através de arestas CHILD.

Definição 6.3.33. Define-se como **tinytree**, um subgrafo contendo apenas um vértice raiz r e todos os vértices e arestas descendentes de r através de qualquer aresta, exceto arestas PROPER CUBE.

Definição 6.3.34. Define-se como **dimensão de uma tinytree T**, o valor correspondente a dimensão do vértice raiz da tinytree T.

Definição 6.3.35. Define-se como **Maxitree**, a tinytree que tem o Maxiroot como vértice raiz. Note que a Maxitree é a única tinytree com dimensão igual a 1.

Definição 6.3.36. Define-se como **subtree**, uma tinytree que cuja dimensão é maior do que 1. Note que a raiz de uma subtree é sempre uma subraiz, logo existe um vértice v que é pai dessa subraiz através de uma aresta CUBE. Diz-se que "**o vértice v é pai de uma subtree T**" ou que "**uma subtree T é filha do vértice v**", se o vértice v é pai da raiz da subtree T.

Definição 6.3.37. Define-se como **supertree** de uma tinytree T, qualquer tinytree que possua um vértice da qual a raiz da tinytree T é descendente através de arestas CUBE. Quando uma tinytree A é supertree de uma subtree B, então pode-se dizer que "a tinytree B é **derivada** de A".

6.3.6 Propriedades construtivas fundamentais

Até este ponto, foram relacionadas diversas propriedades do Grafo que modela um Tinycubes. No entanto, ainda não foi definido como os Contents nos vértices terminais são efetivamente construídos e quais critérios são utilizados para criação das arestas CHILD. Esta seção apresenta definições e propriedades que resolvem essas questões.

6.3.6.1 Arestas CUBE

Propriedade 6.3.6. Propriedade fundamental das arestas CUBE. Para todo vértice regular x, o resultado da aplicação do operador $\mathcal{M}_i^n$ sobre os elementos do conjunto formado pelo Content encontrado em cada um dos vértices terminais descendentes de x utilizando arestas CHILD, tem de ser igual ao Content presente no vértice terminal descendente de x utilizando arestas CUBE.

Propriedade 6.3.7. Arestas **SHARED CUBE** (**I-CUBE**) . Se um vértice x é origem de apenas uma aresta CHILD, e essa aresta aponta para um vértice y, então a aresta CUBE a com origem em x será SHARED CUBE. Além disso, a aresta a apontará para o vértice y, apenas se y for um vértice TOP. Caso contrário, a aresta a apontará para o vértice que é o destino da aresta CUBE com origem no vértice y.

Propriedade 6.3.8. Arestas **PROPER CUBE (X-CUBE)**. Se um vértice x é origem de mais de uma aresta CHILD, então a aresta CUBE a com origem em x será PROPER CUBE. Além disso, a aresta a apontará para um vértice TOP r que não é vértice destino de

nenhuma outra aresta. Note que, com essa definição, vértices destino de arestas PROPER CUBE são sempre vértices raiz de uma subtree, ou subraízes. Além disso, a tinytree que possui o vértice x é uma supertree dessa subtree.

6.3.6.2 Arestas CHILD

Para especificar as regras que controlam a existência das arestas tipo CHILD é necessário realizar mais algumas definições.

Definição 6.3.38. Define-se o conjunto Bool que representa os valores verdadeiro e falso. Matematicamente:

$$Bool = \{Verdadeiro, Falso\} \tag{6.29}$$

Definição 6.3.39. Define-se o predicado ***child***, que retorna Verdadeiro apenas a aresta e for do tipo CHILD. Matematicamente:

$$child(e) \mapsto Bool \quad e \in E \tag{6.30}$$

Definição 6.3.40. Define-se o conjunto $\boldsymbol{E^c}$, subconjunto de E, como o conjunto que contém todas as arestas do tipo CHILD em E. Matematicamente:

$$E^c \subseteq E \wedge (e \in E^c \iff child(e) \wedge e \in E) \tag{6.31}$$

Definição 6.3.41. Define-se a função ***option*** que, ao receber uma aresta CHILD como parâmetro, retorna o valor armazenado nela. Matematicamente:

$$option(e) \mapsto \mathbb{Z} \quad e \in E^c \tag{6.32}$$

Definição 6.3.42. Define-se como $\boldsymbol{Options_{level}}$, o conjunto com todos os valores que uma aresta no nível dimensional *level* pode assumir. Matematicamente:

$$Options_{level} = \{x \in \mathbb{Z}; \exists e \in E^c \wedge L(e) = level \wedge x = option(e)\} \tag{6.33}$$

Definição 6.3.43. Define-se o predicado ***subChild***, que retorna Verdadeiro se o vértice

a é descendente do vértice b através de arestas do tipo CHILD. Matematicamente:

$$subChild(a, b) \mapsto Bool \quad a, b \in V \tag{6.34}$$

Definição 6.3.44. Define-se o conjunto $\boldsymbol{E^M}$, subconjunto de E^c, como o conjunto que contém todas as arestas em E^c que **são descendentes** do vértice Maxiroot através de arestas CHILD. Matematicamente:

$$E^M \subseteq E^c \wedge (e = (u, v) \in E^M \iff \exists u \in V \wedge subChild(u, Maxiroot)) \tag{6.35}$$

Definição 6.3.45. Define-se o conjunto $\boldsymbol{E^N}$, subconjunto de E^c, como o conjunto que contém todas as arestas em E^c que **não são descendentes** do vértice Maxiroot através de arestas CHILD. Matematicamente:

$$E^N = E^c - E^M \tag{6.36}$$

Arestas CHILD descendentes do Maxiroot (Maxitree)

Arestas CHILD não descendentes do Maxiroot (subtrees)

Definição 6.3.46. Define-se $\boldsymbol{V^c_{v,l}}$ como o conjunto dos vértices descendentes de v no nível l através de arestas CHILD. Matematicamente:

$$V^c_{v,l} = \{u \in V; subChild(u, v) \wedge L(v) = l\} \tag{6.37}$$

Definição 6.3.47. Seja r uma subraiz no nível $l_0 = L(r)$. Seja a o vértice pai de r. Nessas condições, define-se $\boldsymbol{V^c_k}$ como o conjunto dos vértices descendentes de a no nível $(l_0 + k)$ através de arestas CHILD. Matematicamente:

$$V^c_k = \begin{cases} V^c_{a,l_0} & k = 0 \\ \{y \in V; \exists e \in E^c \wedge e = (x, y) \wedge x \in V^c_{k-1}\} & k > 0 \end{cases} \tag{6.38}$$

Definição 6.3.48. Define-se $\boldsymbol{E^c_k}$ como o conjunto de arestas CHILD que conectam os vértices descendentes de a através de arestas CHILD que estão no nível $(l_0 + k - 1)$ com os vértices no nível $(l_0 + k)$. Note que E^c_0 não é definido. Matematicamente:

$$E^c_k = \{e \in E^c \wedge e = (x, y) \wedge x \in V^c_{k-1} \wedge y \in V^c_k\} \quad k > 0 \tag{6.39}$$

Definição 6.3.49. Define-se $\boldsymbol{E^c_{k,o}}$ como o subconjunto de E^c_k no qual as arestas tem valor

OPTION igual a o. Matematicamente:

$$E^c_{k,o} = \{e \in E^c_k; option(e) = o\} \quad k > 0 \tag{6.40}$$

Definição 6.3.50. Define-se a notação compacta $\boldsymbol{\alpha_k}$, que representa uma família de caminhos (iniciados no nível l_0) no grafo dimensional, dispostos como uma sequência de possíveis valores OPTION, indo de zero até k. Por exemplo, X_{α_k} com $k = 3$ é uma forma compacta de escrever X_{o_o,o_1,o_2,o_3}, sendo que cada o_i representa todos os valores possíveis de valores OPTION para o nível dimensional $(l_0 + i)$. Pode-se pensar em α_k como representante de todos os caminhos simples possíveis com comprimento k iniciados em algum vértice no nível l_0. Matematicamente:

$$\alpha_k = o_0, ..., o_k \quad o_i \in Options_{(l_0+i)} \wedge i \in [0, k] \tag{6.41}$$

Definição 6.3.51. Define-se $E^c_{\alpha_k}$ como o conjunto de arestas CHILD que tem origem em vértices próprios [5] que são finais de caminhos α_{k-1}. Matematicamente:

$$E^c_{\alpha_k} = \{(x, y) \in E^c \wedge x \in V^P_{\alpha_{k-1}} \wedge y \in V\} \tag{6.42}$$

Definição 6.3.52. Define-se $\boldsymbol{V^S_{\alpha_k}}$ como o conjunto de vértices que são destino de arestas SHARED CHILD no caminho α_k. Matematicamente:

$$V^S_{\alpha_k} = V^S_{o_0,...o_k} = \{y \in V; \exists(x, y) \in E^c_{\alpha_k} \wedge |E^c_{k,o_k}| = 1 \wedge \exists(z, y) \in E^c_{k,o_k}\} \tag{6.43}$$

Definição 6.3.53. Define-se $\boldsymbol{E^S_{\alpha_k}}$ como o conjunto de arestas SHARED CHILD que apontam para os vértices em $V^S_{\alpha_k}$. Matematicamente:

$$E^S_{\alpha_k} = E^S_{o_0,...o_k} = \{e \in E^c_{\alpha_k}; e = (x, y), y \in V^S_{\alpha_k} \wedge option(e) = o_k\} \tag{6.44}$$

Propriedade 6.3.9. Arestas **SHARED CHILD** (em subtrees). Uma aresta a, sub-vértice de uma subraiz r, existe e é do tipo SHARED CHILD se pertence ao conjunto $E^S_{\alpha_k}$.

Definição 6.3.54. Define-se $V^P_{\alpha_k}$ como o conjunto de vértices destino de arestas PROPER CHILD do caminho α_k. Matematicamente:

$$V^P_{\alpha_k} = \begin{cases} V^P_{o_0} = \{r\} & k = 0 \\ V^P_{o_0,...o_k} = \{y \in V; \exists(x, y) \in E^c_{\alpha_k} \wedge |E^c_{k,o_k}| \geq 2\} & k > 0 \end{cases} \tag{6.45}$$

Definição 6.3.55. Define-se $E^P_{\alpha_k}$ como o conjunto de arestas PROPER CHILD do cami-

[5] vértices que são destino de pelo menos uma aresta PROPER

nho α_k. Matematicamente:

$$E^P_{\alpha_k} = \{(x, y) \in E^c_{\alpha_k}; y \in V^P_k\} \tag{6.46}$$

Propriedade 6.3.10. Arestas **PROPER CHILD** (em subtrees). Uma aresta a, sub-vértice de uma subraiz r, existe e é do tipo PROPER CHILD se pertence ao conjunto $E^P_{\alpha_k}$.

6.3.7 Exemplo

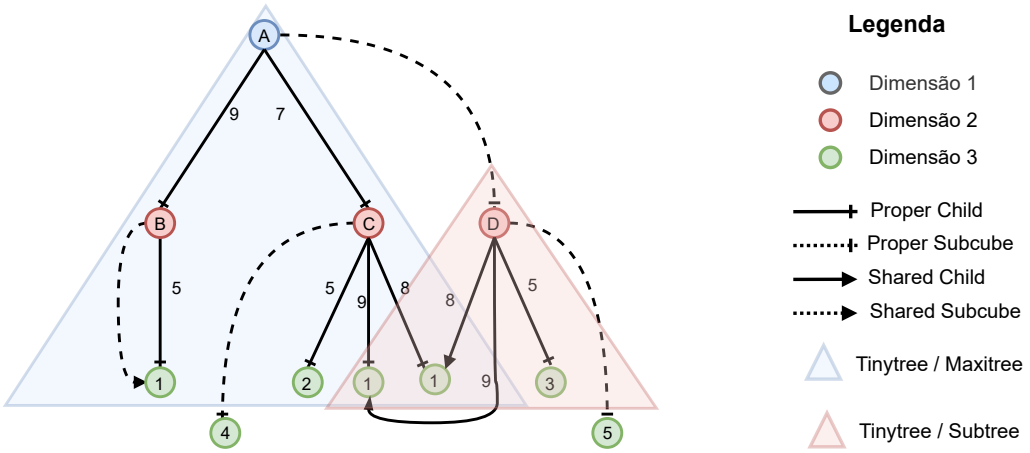


Figura 6.5: Exemplo de tinytrees

A Figura 6.5 exibe um tinycube com informação equivalente ao nanocube da Figura 4.16(b). Ao se comparar as duas figuras, percebe-se que o tinycube é mais compacto do que o nanocube. Isso ocorre porque o tinycube tem sete vértices a menos em decorrência da eliminação das arestas de ligação (ver Seção 4.3). Para melhor compreensão, os vértices regulares do tinycube foram identificados com letras maiúsculas, enquanto os vértices terminais apresentam valores correspondentes a Contents de contagem. Note, inicialmente que os vértices terminais apresentando os valores 4 e 5 não fazem parte de nenhuma tinytree, uma vez que só são apontados por arestas X-CUBE e não são vértices regulares.

Iniciando os comentários descritivos, o vértice A é o Maxiroot, que é a raiz (root) da Maxitree. O Maxiroot é um vértice regular na dimensão 1 do tipo TOP (sem arestas incidentes da mesma dimensão). O Maxiroot é único vértice do tinycube que não pode ser removido e que não tem arestas incidentes.

Saindo do Maxiroot através da aresta PROPER CUBE (X-CUBE, EXTERNAL TOP por criar uma nova tinytree), é possível atingir o vértice D na dimensão 2. Este vértice

é um TOP e uma subraiz. O vértice D tem o vértice A (Maxiroot) como pai. Por ser uma subraiz, este vértice é a base de uma tinytree na dimensão 2, que é uma subtree por não ser a Maxitree. O vértice pai desta tinytree/subtree é o vértice A, que é o Maxiroot. O vértice D possui três arestas CHILD, duas PROPER CHILD e uma aresta SHARED CHILD com valor 9. A aresta SHARED CHILD aponta para um vértice terminal (que é TOP e está na dimensão 3) de uma supertree, cuja a raiz é o Maxiroot, logo a supertree é a Maxitree.

Saindo do vértice A (Maxiroot), à partir da aresta PROPER CHILD com valor 9, atinge-se o vértice B. O vértice B é TOP e está na dimensão 2. Como só possuí uma aresta CHILD, sua aresta CUBE é do tipo SHARED CUBE (I-CUBE, INTERNAL TOP por estar na mesma tinytree), e aponta para o mesmo vértice terminal TOP(3) que a aresta com valor 5 filha de B, aponta.

Saindo do vértice A (Maxiroot), à partir da aresta PROPER CHILD com valor 7, atinge-se o vértice C. O vértice C é TOP e está na dimensão 2. Como possuí três arestas CHILD, sua aresta CUBE é do tipo PROPER CUBE (X-CUBE), e aponta para um vértice terminal TOP(3), que tem de armazenar o mesmo Content que o Content resultante do "merge" de todos os vértices filhos de C.

6.4 Algoritmos de Inserção e de Remoção

Os algoritmos de inserção e de remoção apresentados nesta seção são responsáveis por manter a consistência do índice do Tinycubes e, à partir de um endereço gerado por uma função externa aos algoritmos, selecionar os vértices terminais adequados que devem ser atualizados a cada operação. As operações para manipulação dos Contents presentes nos vértices terminais também são realizadas através de funções externas, sendo apenas executadas (chamadas) pelos algoritmos. A utilização de funções externas oferece independência de contexto para ambos algoritmos visto que seu correto funcionamento independe, tanto dos dados presentes nos records, quanto das informações que devem ser extraídas.

Ambos algoritmos baseiam-se nos mesmos princípios e procedimentos gerais listados a seguir.

- Os algoritmos percebem um Tinycubes como uma tinytree primária (Maxitree) que pode apontar para subtrees que, por sua vez, podem recursivamente apontar para outras subtrees como ilustrado na Figura 6.6;

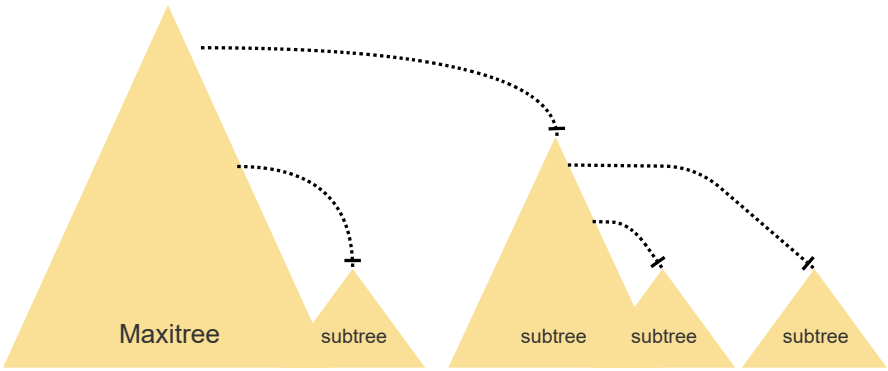


Figura 6.6: Maxitree e subtrees

- Inicialmente, o algoritmo que realiza a operação (inserção ou remoção) deve descer pela Maxitree até o vértice terminal, eventualmente criando arestas e vértices pelo caminho, registrando quais vértices foram visitados, para, em seguida, realizar a operação sobre o vértice terminal visitado;

- Após processar o vértice terminal, o algoritmo retorna pelos mesmos vértices visitados, em ordem inversa da ordem utilizada para descida, ajustando as arestas CUBE desses vértices. Esses ajustes podem levar o algoritmo a realizar a mesma operação em curso (inserção ou remoção) na subtree filha do vértice sendo ajustado, caso ela exista.

- A execução da operação (inserção ou remoção) sobre uma subtree é similar a da Maxitree, à exceção de que, na descida em uma subtree, quando uma aresta SHARED CHILD é encontrada[6], em alguns casos detalhados adiante, o algoritmo interrompe a descida sem chegar ao vértice terminal[7]. Em qualquer caso, finda a descida, o algoritmo volta realizando ajustes como no caso da Maxitree.

Nas seções a seguir, cada um dos algoritmos terá suas peculiaridades evidenciadas e será explicado com maior riqueza de detalhes.

6.4.1 Algoritmo de Inserção

O algoritmo de inserção do Tinycubes pode ser entendido como um processo realizado em duas partes. Na primeira parte, desce-se pela Maxitree até atingir o vértice terminal determinado pelo endereço informado e, em seguida, faz-se a atualização de todos

[6]Note que não existem arestas SHARED CHILD na Maxitree porque este tipo de aresta indica que uma subtree está utilizando a estrutura de uma supertree e não existe uma supertree para a Maxitree.

[7]Os resultados dos experimentos realizados indicam que isso pode ocasionar uma redução significativa tando no tempo de execução quanto no consumo de memória.

os Contents à partir dos dados presentes no record. Na segunda parte, possivelmente recursiva, as subtrees cujos vértices pai foram visitados anteriormente, são processadas e as atualizações necessárias nos Contents dos vértices terminais são efetuadas, causando, eventualmente, inserções em outras subtrees utilizando subendereços.

O algoritmo considera que, se durante a inserção na Maxitree, a única alteração ocorreu num vértice terminal, não tendo sido necessária a criação de nenhuma aresta, então não será necessário criar nenhuma nova aresta em qualquer subtree do tinycube. O processo de inserção do record nas subtrees (utilizando os subendereços apropriados) consistirá apenas na atualização dos vértices terminais necessários para que a integridade dos Contents seja mantida.

Por outro lado, se durante a inserção na Maxitree forem criados vértices e arestas, certamente será necessário criar pelo menos uma aresta CHILD e eventualmente também será necessário criar um ou mais vértices em algumas das subtrees filhas de vértices visitados anteriormente. Nesse caso, para evitar a proliferação de vértices e arestas, o algoritmo sempre tentará criar uma aresta SHARED CHILD com origem na subtree sendo atualizada e destino numa supertree[8]. Porém, isso nem sempre é possível, e nesse caso, o algoritmo criará uma nova aresta PROPER CHILD, o vértice destino necessário para essa nova aresta e fará as todas as atualizações necessárias para manter estrutura consistente com as propriedades especificadas para um Tinycubes.

Algorithm 6.1: Algoritmo de inserção para o Tinycubes

```
1 proc tinycubesInsert(Maxiroot, record, schema)
2     address ← createAddressFromRecord(record, schema)
3     memories ← array [schema.nDimensions+1] of {op, nodes[schema.nLevels+1]})
4     memory ← memories[0]
5     visited ← nodesCreate(); updated ← nodesCreate()
6     insertIntoTree(Maxiroot, 0, record, address, memory, null, visited, updated, schema)
7     nodesDestroy(updated); nodesDestroy(visited)
```

Algorithm 6.2: Inserção de um record numa tinytree

```
1 proc insertIntoTree(root, iroot, record, address, mem, prev, visited, updated, schema)
2     inode, child ← iGoDown(root, iroot, record, address, mem, prev, visited, updated, schema)
3     iGoBack(mem, record, address, iroot, inode, child, visited, updated, schema)
```

Quando a tinytree é a Maxitree, o algoritmo de descida limita-se a seguir pelo caminho

[8]Como uma subtree só é processada para ajustar as arestas CUBE durante a fase de subida de uma supertree, é correto afirmar que a supertree que possui o vértice para o qual a aresta SHARED CHILD vai apontar, está sendo processada

Algorithm 6.3: Algoritmo de descida para inserção de um record

```
1  proc iGoDown(root, iroot, record, address, mem, prev, visited, updated, schema)
2      node ← root; inode ← iroot
3      ichild ← inode + 1; alreadyUpdated ← false; bottom ← false
4      loop
5          option ← address[inode]
6          child, edge, isShared ← locateChild(node, option)
7          if isNull(child):
8              if isNull(prev):                                    // or prev.op = NONE
9                  child ← createNode()
10                 createChildEdge(node, child, PROPER, option)
11                 if mem.op = NONE: mem.op ← NEW
12             else: // reuse node from prev mem
13                 child ← prev.nodes[ichild]
14                 createChildEdge(node, child, SHARED, option)
15                 completeNodesAfterIndex(mem.nodes, prev.nodes, ichild, schema)
16                 mem.nodes[ichild] ← child
17                 mem.op ← prev.op
18                 break // stops descent
19         elseif isShared:
20             if prev.op = NONE:                          // No insertion in prev mem?
21                 completeNodesAfterIndex(mem.nodes, prev.nodes, ichild, schema)
22                 break // stops descent
23             elseif child = prev.nodes[ichild]:                // Is this 'child' recent?
24                 mem.op ← prev.op
25                 completeNodesAfterIndex(mem.nodes, prev.nodes, ichild, schema)
26                 break // stops descent
27             else:
28                 if isTerminal(ichild, schema) and nodesLocated(updated,child):
29                     alreadyUpdated ← true
30                 child ← createProperChildForEdge(edge)
31                 if mem.op = NONE:  mem.op ← DESHARE
32         mem.nodes[ichild] ← child
33         if isTerminal(ichild, schema):
34             bottom ← true
35             break
36         node ← child; inode ← ichild; ichild ← inode + 1
37     if bottom and not alreadyUpdated:
38         terminalInsert(child, record, schema)
39         nodesInsert(updated, child)
40     return inode, child
```

determinado pelo endereço, registrando os vértices visitados num array (nodes) para uso posterior, até: ou chegar a um vértice terminal, atualizá-lo e retornar; ou não encontrar uma aresta CHILD com o valor especificado pelo endereço. No último caso, o algoritmo começa a criar as arestas PROPER CHILD com os valores determinados pelo endereço bem como os respectivos vértices destino até criar um vértice terminal, atualizá-lo e retornar. No processo, a função memoriza a informação de que houve uma alteração na tinytree (linha 11, $mem.op \leftarrow NEW$).

Quando a tinytree não é a Maxitree, existem muito mais detalhes a se considerar. O algoritmo de descida começa a descer à partir da subraiz procurando por arestas CHILD que correspondem aos valores do sub-endereço sendo considerado. Enquanto encontrar arestas PROPER CHILD, o algoritmo apenas continua descendo até chegar ao vértice terminal, atualizá-lo e retornar.

Caso não encontre (linha 7, *isNull(Child)* é verdadeiro) nenhuma aresta com o valor desejado, então necessariamente houve uma alteração na supertree da dimensão imediatamente anterior a esta tinytree. Nesse caso, o registro do caminho percorrido para essa supertree é utilizado para identificar qual novo vértice foi criado. A partir dessa informação, o algoritmo cria uma aresta SHARED CHILD com origem no último vértice que ela localizou na descida e destino no vértice do nível correto que foi recém criado na supertree. Como existe a possibilidade de que esta tinytree venha a ser utilizada como supertree no processo recursivo, o algoritmo completa o registro do caminho percorrido desta tinytree com o registro da parte final do caminho percorrido na supertree e registra que houve uma modificação nesta tinytree.

Caso o algoritmo encontre uma aresta SHARED CHILD com o valor procurado, passam a existir três cenários. No primeiro, não houve alteração na supertree, logo não é necessário fazer mais nada e o algoritmo só completa o seu caminho com o caminho da supertree. No segundo cenário, houve alteração na supertree mas foi numa aresta num nível mais baixo da tinytree do que a aresta SHARED CHILD, logo só é necessário completar o caminho com os dados da supertree e retornar. No terceiro caso, é necessário criar uma aresta PROPER CHILD à partir da aresta SHARED CHILD. Isso implica em criar um novo vértice destino z para esta nova aresta e também duplicar cada aresta CHILD do vértice v que era destino da aresta SHARED CHILD, fazendo cada nova aresta duplicada SHARED CHILD, mantendo o destino nos mesmos vértices das arestas originais e fazendo a origem ser o novo vértice destino z. Além disso, a aresta CUBE do vértice v é duplicado como SHARED CUBE, mantendo o mesmo vértice destino original e fazendo a origem

ser o novo vértice z.

Algorithm 6.4: Algoritmo de subida durante a inserção de um record

```
1  proc iGoBack(mem, record, address, iroot, inode, child, visited, updated, schema)
2      while inode >= iroot:
3          node ← mem.nodes[inode]
4          content, icontent, sharedContent ← getCubietInfo(node)
5          mustUpdateContent ← false
6          if hasSingleChild(node):
7              old ← content
8              if icontent = inode + 1:
9                  content ← child
10             else:
11                 content, icontent, dummy ← getCubietInfo(child)
12             if old ≠ content or not sharedContent:
13                 setContentInfo(node, content, SHARED)
14         elseif sharedContent:
15             mustUpdateContent ← not nodesLocated(visited, content)
16             content ← shallowCopy(content)
17             setContentInfo(node, content, PROPER)
18         else:
19             mustUpdateContent ← true
20         if mustUpdateContent:
21             if isTerminal(icontent, schema):
22                 terminalInsert(content, record, schema)
23             else:
24                 insertIntoTree(content, icontent, record, address, nextMemory(mem), mem,
                     visited, updated, schema)
25                 nodesInsert(visited, content)
26         child ← node; inode ← inode - 1;
```

6.4.2 Algoritmo de Remoção

Inicialmente deve-se ressaltar que cada execução do algoritmo de remoção não tem que realizar a remoção de algum objeto da estrutura. A remoção de um record consiste, de fato, em desfazer os efeitos lógicos da uma inserção anterior desse mesmo record. Dito de outra forma, após a remoção de um record, as informações produzidas pela estrutura não devem mais refletir uma antiga inserção desse record. Isso significa que, se um mesmo record for inserido duas vezes através do algoritmo de inserção, então será necessário executar o algoritmo de remoção duas vezes utilizando o mesmo record a fim de que um Tinycubes deixe de produzir informações considerando as duas inserções realizadas.

Tal como o algoritmo de inserção, o algoritmo de remoção do Tinycubes pode ser entendido como um processo realizado em duas partes. Na primeira parte, o algoritmo desce pela Maxitree até localizar o vértice terminal especificado pelo endereço fornecido para, em seguida, desfazer os efeitos da inserção do record sobre cada Content presente

nesse vértice terminal. Existem dois cenários possíveis à partir deste ponto. No primeiro cenário, o vértice terminal ainda armazena Contents afetados por alguma inserção passada, portanto tanto esses Contents quanto o próprio vértice devem ser mantidos. No segundo cenário, é detectado que os Contents do vértice terminal não possuem mais valores afetados por inserções passadas, portanto o vértice terminal e seus Contents devem ser removidos.

No primeiro cenário, no qual fica claro que o vértice terminal da Maxitree deve ser mantido, o algoritmo registra que não houve remoção e passa visitar subtrees filhas dos vértices visitados aplicando o mesmo algoritmo de remoção, com parâmetros adequadamente ajustados. Embora não esteja refletido no algoritmo em si, sabe-se que nesse cenário, não haverá remoção de nenhum vértice em qualquer subtree pois não houve alteração na Maxitree. Na prática, o que ocorre é o ajuste de todos os Contents relevantes nas subtrees de modo a refletir a remoção do record na Maxitree.

Ja no segundo cenário, no qual o vértice terminal da Maxitree foi realmente removido, é necessário retornar pela tinytree verificando se a variação na quantidade de arestas CHILD do vértice pai do vértice terminal, digamos p pode causar algum efeito adicional. O processo descrito a seguir inicia-se com o vértice p da Maxitree subindo até a sua raiz, mas é recorrente em todas as tinytrees que são visitadas durante uma remoção. A remoção de uma aresta CHILD de um vértice x pode exigir uma alteração na aresta CUBE deste mesmo vértice x, ou mesmo a remoção do próprio x. Existem três possibilidades após a remoção de uma aresta CHILD filha de x: (i) O vértice x ficou sem filhos, logo deve ser removido e isso deve ser sinalizado para o seu vértice pai; O vértice x ficou com mais de 2 filhos, portanto basta atualizar sua subtree recursivamente; (iii) O vértice x passou a ter apenas um filho, logo sua aresta CUBE mudará de PROPER (eram dois filhos) para SHARED (restou apenas um) e toda a sua subtree deve ser descartada.

A maior complexidade do algoritmo está justamente no descarte da subtree de um vértice que ficou com apenas um filho após a remoção de uma aresta PROPER CHILD. A dificuldade se origina nas propriedades das arestas SHARED CHILD devido as duas formas em que são utilizadas: quando a aresta aponta para o vértice de uma supertree (a subtree reusou parte da estrutura de outra tinytree) e quando a aresta aponta para um vértice desta subtree (um vértice desta subtree é usado por outra subtree). Um algoritmo complementar chamado "purge" foi desenvolvido para superar esses desafios, removendo os vértices da subtree que podem ser realmente descartados.

Quando o algoritmo de descida analisa uma subtree, é possível que o endereço for-

necido atinja uma aresta SHARED CHILD. Nesse caso, o algoritmo verifica se o vértice destino dessa aresta foi descartado como parte dessa operação de remoção ou não. Se vértice já foi descartado durante esse processo, então o algoritmo remove essa aresta, sinaliza que houve remoção na subtree e retorna. Se o vértice destino não foi removido, então o algoritmo apenas retorna. Em ambos os casos, a descida é interrompida sem atingir o vértice terminal.

Note que o algoritmo de descida necessita saber se um vértice destino de aresta SHARED CHILD foi já descartado durante a remoção, o que sugere a preservação dos vértices descartados até que todo o processo seja concluído, apenas marcando-os como descartados[9].

Algorithm 6.5: Algoritmo de remoção para o Tinycubes

```
1 proc tinycubesRemove(Maxiroot, record, schema)
2     address ← createAddressFromRecord(record, schema)
3     memories ← array [schema.nDimensions+1] of {op, nodes[schema.nLevels + 1]}
4     mem ← memories[0]
5     roots ← rootsCreate()
6     visited ← pointersCreate(256)
7     removeFromTree(Maxiroot, 0, record, address, mem, visited, roots, schema)
8     pointersDestroy(visited)
9     rootsDestroy(roots)
```

Algorithm 6.6: Remoção numa tinytree

```
1 proc removeFromTree(root, iroot, record, address, mem, visited, roots, schema)
2     edges ← array [schema.nValues]
3     inode, child, deleting ← rGoDown(root, iroot, record, address, mem, visited, edges,
        schema)
4     if isNull(child): return // Address not found
5     rGoBack(mem, record, adddress, iroot, inode, child, deleting, visited, roots, schema)
```

[9]Alternativamente seria possível remover o vértice imediatamente e registrar que ele foi removido numa estrutura acessória, mas isso exigiria que qualquer acesso a um vértice fosse precedido de um teste para verificar se esse vértice ainda está na estrutura ou não, o que tornaria todo o processo de remoção mais lento

Algorithm 6.7: Remoção - descida na tinytree

```
1  proc rGoDown(root, iroot, record, address, mem, visited, edges, schema)
2      node ← root; inode ← iroot; ichild ← inode + 1
3      loop
4          option ← address[inode]
5          child, edges[inode], isShared ← locateChild(node, option)
6          if isNull(child): return 0, null, false
7          if isShared:
8              if headerDeleted(child): deleting← true
9              break // stops descent
10         mem.nodes[ichild] ← child
11         if isTerminal(ichild, schema): break
12         node ← child; inode ← ichild; ichild ← inode + 1
13     if isTerminal(ichild, schema):
14         count ← terminalRemove(child, record, schema)
15         if count = 0:
16             deleteNode(child)
17             deleting← true
18     return inode, child, deleting;
```

Algorithm 6.8: Remoção - retorno da subida ajustando a tinytree

```
1  proc rGoBack(mem, record, address, iroot, inode, child, deleting, visited, roots,
     schema)
2      while inode >= iroot:
3          node ← mem.nodes[inode]
4          content, icontent, sharedContent ← getCubietInfo(node)
5          if deleting: removeChildEdge(node, edges[inode])
6          nChildren ← getChildrenCount(node);
7          if nChildren>1:
8              if not sharedContent or not nodesLocated(visited, content):
9                  if isTerminal(icontent, schema):
10                     terminalRemove(content, record, schema)
11                 else:
12                     removeFromTree(content, icontent, record, address, next(mem),
                         visited, roots, schema)
13                     pointersInsert(visited, content);
14             deleting ← false
15         elseif nChildren=0:
16             releaseContent(node)
17             if inode = iroot: return // Did it reach the root? Stop
18             deleteNode(node); deleting← true
19         elseif not deleting:
20             if icontent = inode + 1:
21                 content = child;
22             else:
23                 content, dummy ← getCubietInfo(node)
24             setContentInfo(node, content, SHARED)
25         else: // only one child left
26             if not sharedContent:
27                 if isTerminal(icontent, schema):
28                     terminalRemove(content, record, schema);
29                 else:
30                     removeFromTree(content, icontent, record, address,
                         nextMemory(mem), visited, roots, schema);
31             rootsdAdd(roots, node, content, icontent, sharedContent);
32             child ← getFirstChild(node);
33             if inode + 1 = icontent:
34                 content = child;
35             else:
36                 content, dummy1, dummy2 ← getCubietInfo(child)
37             setContentInfo(node, content, SHARED)
38             deleting ← false;
39         child ← node; inode ← inode - 1;
```

Capítulo 7

Tinycubes - Modularização

A tecnologia Tinycubes é baseada em módulos, componentes de software contendo algoritmos e/ou estruturas de dados, utilizados na construção de objetos tinycubes e/ou no processo de consulta e extração de informações. De fato, a maioria das operações essenciais para o funcionamento da tecnologia foram implementadas utilizando módulos. Assim, o usuário da tecnologia pode selecionar dentre as versões alternativas das funcionalidades existentes, aquelas que melhor atendem suas necessidades. Como exemplo, pode-se ter implementações diferentes para uma mesma funcionalidade, na qual uma delas é otimizada para velocidade mas limitada a dados estáticos enquanto outra permite a inserção e remoção de novos dados durante a operação.

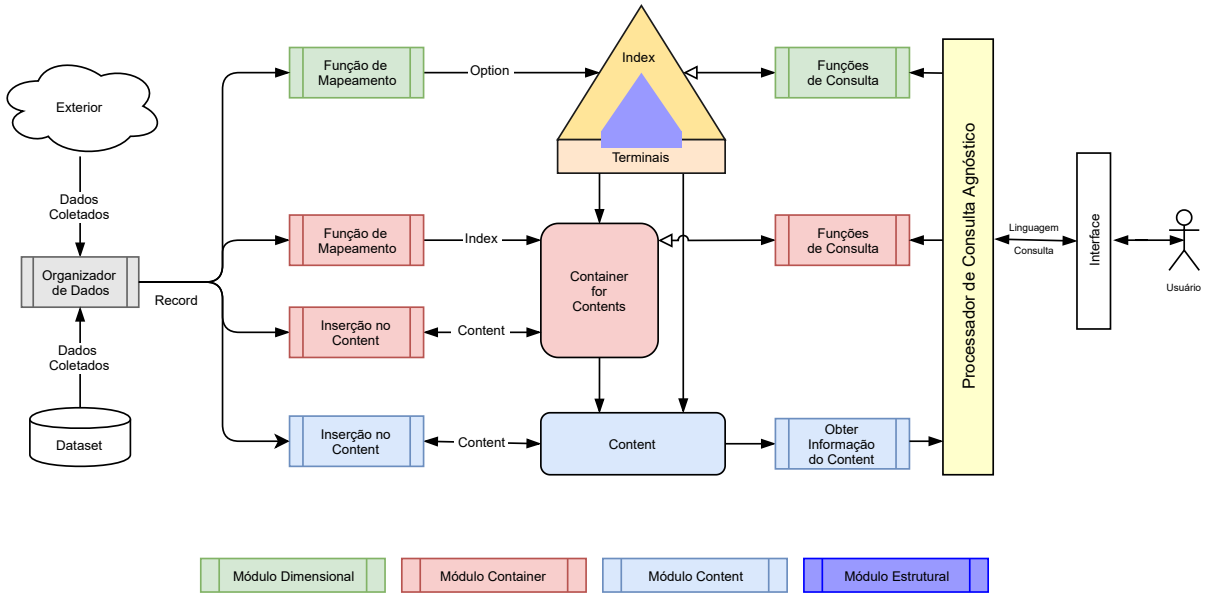


Figura 7.1: Oportunidades de modularização

Esta Seção descreve o sistema de modularização definido pela tecnologia Tinycubes e bem como uma visão sucinta da forma como os módulos são instalados na tecnologia.

Por estarem vinculados a implementação, diversas características do sistema de módulos serão apresentadas através de trechos comentados de código na linguagem C.

7.1 Introdução

A Figura 7.1 ilustra um diagrama esquemático da tecnologia Tinycubes, evidenciando através de cores, os procedimentos executados durante o recebimento dos dados e a subsequente produção de informação que podem ser implementados como módulos ou como biblioteca de código como no procedimento "Organizador de dados", que por esta razão não será analisado aqui.

A Figura 7.2 exibe um diagrama esquemático dos módulos definidos pela tecnologia. Com se pode observar, vários módulos são, na realidade a implementação de uma única classe, como os Módulos Dimensionais (classe "Classe") e Módulos Content (classe "Content"). Já o módulo Container implementa duas classes ("Classe" e "Content"). Por outro lado os módulos estruturais não implementam nenhuma classe.

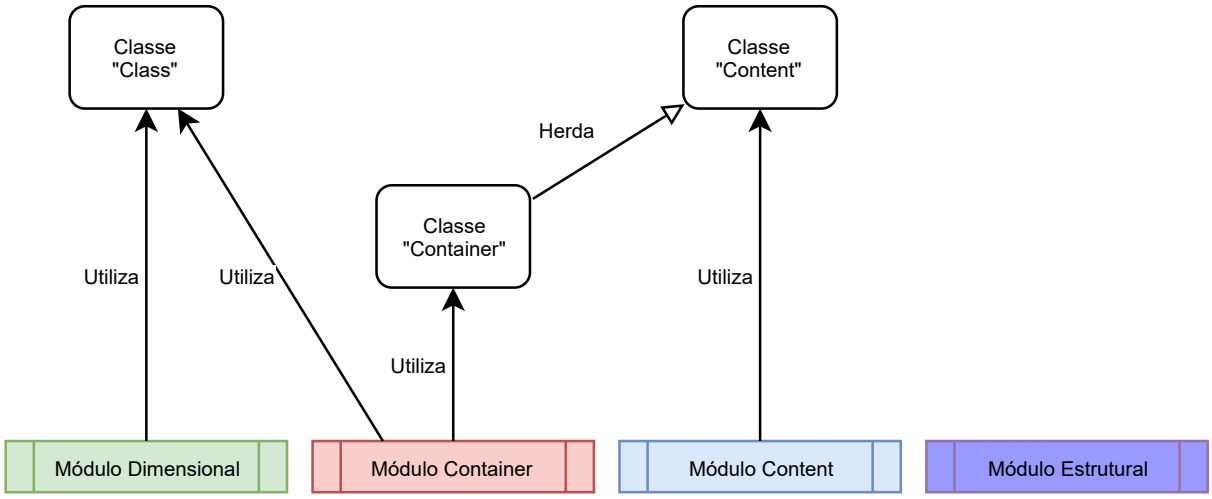


Figura 7.2: Diagrama esquemático do sistema de módulos

Os tipos de módulo suportados por um Tinycubes são:

Módulo Dimensional é responsável por prover valores a serem armazenados numa dimensão e também operações, funções que utilizarão esses valores durante a consulta e na recuperação de informação.

Módulo Content é responsável por implementar os Contents que são utilizados na produção da informação.

Módulo Container (Módulo de Classe + Content) é ao mesmo tempo responsável por implementar uma dimensão fora do sistema de índice, como um módulo dimensional, e por dar acesso à Contents que produzirão as informações resultantes de consultas.

Módulo Estrutural é responsável por implementar uma camada abstrata que dá acesso aos vértices e arestas do Tinycubes, possibilitando que o algoritmo possa ser fisicamente implementado utilizando uma abordagem diferente da tradicional abordagem via ponteiros.

7.2 Módulos Dimensionais

Módulos dimensionais são os elementos da tecnologia que implementam a classe "Classe", componentes de software que fornecem tanto funções de mapeamento 6.2.2 quanto as funções que implementam os operadores usados numa consulta para selecionar os elementos necessários para extração de informação. Os Módulos dimensionais são o produto final da atividade de pesquisa *AP6*. Existem três tipos de Classe, cada uma atendendo a diferentes finalidades. São elas:

VALUE Esse tipo de classe é utilizada na criação e utilização das arestas referentes a um único nível do Tinycubes. Deve fornecer a função de mapeamento que determina o valor da aresta de um nível do Tinycubes. Essa classe também define funções utilizadas durante as consultas para selecionar ou não uma aresta.

Um exemplo típico é a classe "cat", cuja a função de mapeamento converte parte de um Record em um valor inteiro e oferece as operações "eq" (igual) e "neq", que possibilitam, respectivamente, identificar se um valor é igual ou diferente de um ou mais valores especificados na consulta.

DIMENSIONAL É utilizada na criação e utilização das arestas de todos os níveis de uma dimensão do Tinycubes. Está associada a dimensões complexas, que são tratadas como um todo por esta classe. Deve fornecer uma função de mapeamento que determina de uma só vez, o valor de todas as aresta correspondentes à dimensão. Esse tipo de classe também deve definir funções que podem ser utilizadas nas consultas sobre uma dimensão como um todo.

Um exemplo típico é a classe "geo", cuja a função de mapeamento converte uma coordenada no formato latitude, longitude numa sequência de valores inteiros compatíveis com uma Quadtree que é utilizada para preencher todos os níveis de uma

dimensão. Essa classe oferece ainda diversas operações para seleção de arestas durante a consulta, como a operação "zrect", que identifica se um caminho dimensional está dentro ou fora de um retângulo definido por dois pares de coordenadas e um nível de zoom da Quadtree.

CONTAINER É utilizada na criação e utilização na dimensão implementada por um Container e será discutida na Seção 7.4.

Cada Classe tem especificar vários parâmetros necessários para criação das arestas CHILD do índice, bem como definir os operadores que serão utilizados para selecionar ou não as arestas criadas pela Classe durante uma consulta. A seguir serão listados alguns dos parâmetros mais relevantes utilizados na definição da Classe[1]. Para ver a lista completa de parâmetros, consulte as listagem comentadas a seguir.

name identifica a classe, servindo como nome do construtor da dimensão durante a especificação do esquema

class_type identifica o tipo de classe

to_address rotina que converte partes de um Record na parte do endereço correspondente a um ou mais níveis da dimensão

get_distinct_info rotina que informa uma aproximação do número de valores distintos da (parte da) dimensão associada à classe

As listagens 7.1 e 7.2 apresentam um exemplo completo de instalação de uma Classe. A Listagem 7.1 apresenta o código fonte comentado utilizado para a instalação da classe "geo". Na listagem, vê-se que essa classe será registrada com o nome "geo" (linha 5) e definida como uma classe DIMENSIONAL (linha 8), portanto deve definir todos os valores de aresta de uma dimensão.

Como ocorre em todas as classes, é necessário informar à tecnologia detalhes para execução da operação "group-by". Operações de "group-by" são as operações tipicamente utilizadas para criar histogramas. Como exemplo do funcionamento dessa operação, considere uma tabela com as colunas "estado", "cidade", "número de habitantes". Uma operação de "group-by" utilizando como localizador a coluna "estado" e como valor resultante o "número de habitantes" produziria como resultado final uma relação com duas colunas, uma

[1] Os nomes dos parâmetros estão em inglês porque correspondem a elementos de programas em C, que não aceita acentos

delas contendo a informação do "estado" e na outra a agregação (total) do "número de habitantes" por "estado". Resumindo, uma operação "group-by" cria grupos de informações identificados por chaves, compostas pelo(s) valor(es) utilizado(s) para localizar as informações do resultado, e atribui para a cada grupo, o Content combinado ("merge") de todas informações do resultado que foram localizadas utilizando a chave que identifica o grupo.

A Listagem 7.1 contém alguns parâmetros importantes para configuração da operação "group-by". Na linha 17, informa-se que o "group-by" para elementos desta classe terá uma chave composta por dois valores. No caso específico desta classe, serão as coordenadas geográficas no padrão Tile(Seção 2.6) do Content retornado. Na linha 22, informa-se à tecnologia se a dimensão que serve como chave em um "group-by" implementado pela classe exige que exista uma operação de seleção sobre esta mesma dimensão ou não. Isso é importante, especialmente para classes dimensionais, porque um "group-by" realizado sobre as localidades de um mapa, poderia gerar, hipoteticamente, um grupo para cada 3 metros quadrados de território sendo analisado, ou seja, seria demasiadamente excessivo. A linha 26 define como é implementada a função "get_distinct_info", que informa o número de valores distintos presentes na dimensão e que serão utilizados para criar os grupos resultantes.

Listagem 7.1: Registro da classe geo

```
1  void register_class_geo(void) {
2  static Class c;
3
4          // nome da classe e do construtor
5          c.name="geo";
6
7          // indica que o construtor cria uma dimensao completa
8          c.class_type = DIMENSIONAL_CLASS;
9
10         // registro da funcao mapeadora
11         c.to_address = rec_lat_lon_to_address;
12         c.to_address_types = strdup("DD"); //DoubleDouble
13         c.to_address_fmt = strdup("FF"); //FieldField
14
15         // indica que cada indice utilizado para identificar
16         // os elementos num group-by e composto por 2 valores
17         c.n_group_by_indexes=2;
18
19         // indica que se uma dimensao desta classe for utilizada
20         // num group-by, e necessaria uma clausula 'where'
21         // mencionando a dimensao para evitar excessso de dados
22         c.group_by_requires_rule=1;
23
24         // funcao que indica quantos itens distintos tem num resultado
```

```
25          // usado para dimensionar o array do group_by
26          c.get_distinct_info=&get_distinct_info;
27
28          // funcoes para criar e destruir uma area de trabalho
29          // para que as operacoes acionadas pel owhere possam
30          // ser executadas
31          c.create_op_data=&create_op_data;
32          c.destroy_op_data=&destroy_op_data;
33
34          // operador obrigatorio que sempre retorna true
35          c.op_all_plain=op_all;
36          c.op_all_group_by=NULL;//lat_lon_op_all_group_by;
37
38          //registra a classe no sistema de modulos
39          register_class(&c);
40
41          // chama a rotina que registra os operadores da classe
42          register_ops();
43  }
```

A Listagem 7.2 apresenta o código fonte comentado utilizado para instalação dos operadores da classe "geo". Operadores são funções utilizadas pelas consultas para decidir se uma determinada aresta deve ou não ser percorrida no esforço de localização dos vértices terminais que farão parte da consulta. Nessa listagem estão sendo registrados dois operadores, "zrect" (linhas 3 até 15) e "zpoly" (linhas 18 até 27). O registro de cada operador é realizado preenchendo-se uma estrutura de dados que contém os parâmetros utilizados pela tecnologia. Destacam-se os parâmetros "op" (linhas 6 e 21) e "op_group_by" que indicam quais funções criadas pelo módulo implementarão: (i) a rotina que realiza uma seleção simples das arestas (parâmetro "op") e (ii) a rotina que seleciona arestas para execução de um "group-by".

Outros parâmetros interessantes referem-se a utilização dessas funções pela linguagem de consulta. Os parâmetros "min_args" e "max_args" quantos valores podem ser passados para o operador na consulta. Enquanto a operação "zrect" determina 5 valores (linhas 10 e 11) sugerindo o formato "zrect zoom, lat_1, lon_1, lat_2, lon_2", a operação "zpoly" determina um mínimo de 5 valores (linha 21) mas um máximo de 101 valores (linha 22), sugerindo o formato "zpoly zoom, lat_1, lon_1, lat_2, lon_2, ... lat_{100}, lon_{100}"

Listagem 7.2: Registro das operações para classe geo

```
1  static void register_ops(char* classname) {
2          //registra o operador 'zrect'
3          static QOpInfo qoi_zrect;
4
5          //———— registra o operador 'zrect'
6          qoi_zrect.op = op_zrect; // simples where
7          qoi_zrect.op_group_by = op_zrect_group_by; //em group-bys
```

```
8
9         // descreve como sao os argumentos do operador
10        qoi_zrect.min_args = 5;
11        qoi_zrect.max_args = 5;
12        qoi_zrect.fmt = "IDDDd";
13
14        //registra a funcao como membro da classe 'geo'
15        register_op(classname,S,&qoi_zrect);
16
17        //-------registra o operador 'zpoly'
18        static QOpInfo qoi_zpoly;
19
20        // registra a operacao zpoly quando utilizadas num group-by
21        qoi_zpoly.op = op_zpoly; // simpleswhere
22        qoi_zpoly.op_group_by = op_zrect_group_by; //emgroup-bys
23
24        //descreve como sao os argumentos do operador
25        qoi_zpoly.min_args = 5;
26        qoi_zpoly.max_args = 101;
27        qoi_zpoly.fmt = "IDDDd";
28
29        //registra a funcao como membro da classe 'geo'
30        register_op(classname, S, &qoi_zpoly);
31 }
```

7.3 Módulos Content

Os Módulos Content implementam uma instância da classe Content, tendo como propósito fornecer as funções de armazenamento de valores e extração de informação de Contents. Os módulos Content são o produto final da atividade de pesquisa *AP7*.

A Listagem 7.3 ilustra como a tecnologia implementa os valores a serem armazenados num Content que serão utilizados posteriormente na produção de informações de média (Avg). Vê-se que cada instância do Content (definição 6.2.3.1) armazena dois números: "sum" (linha 2), responsável por acumular o somatório do dado selecionado nos Records recebidos (conjunto Suíte - Seção 6.2.3.2) e o valor "n" (linha 3), responsável por contar quantas somas (amostras) o campo "sum" está acumulando.

Listagem 7.3: Content Avg (average - media)

```
1 typedef struct {
2         double sum; // armazena a soma dos valores
3         int    n;   // armazena a contagem de valores
4 } AvgContent;
```

A Listagem 7.4 exibe o código C que demonstra como o Content "AvgContent" é regis-

trado na tecnologia. Existem diversas categorias de parâmetros que devem ser fornecidos para um registro correto. Destacam-se: parâmetros que descrevem os "parâmetros" que cada Content aceita durante sua configuração (linhas 5 a 9), parâmetros para utilização das funções "merge"(Seção 6.2.3.6) e "extract" [Seção 6.2.3.3] (linhas 11 a 18) e parâmetros que especificam cada uma das funções que manipulam os Contents(Seção 6.2.3) (linhas 21 a 34), a saber: "parse_param", "insert", "remove", "merge", "result".

Listagem 7.4: Registro do Content Avg

```
1  static void avg_register(void) {
2          static ContentInfo ci;
3          memset(&ci,0,sizeof(ContentInfo)); // inicializacao - coloca C0 no content
4
5          // parametros
6          ci.n_param_fields = 1; // num de campos do record utilizados
7          ci.min_params = 0; // num minimo de parametros adicionais
8          ci.max_params = 0; // num maximo de parametros adicionais
9          ci.params_size = 0; // bytes para armazenar parametros adicionais
10
11         // bytes necessarios para armazenar valores do content
12         ci.private_size = sizeof(AvgContent);
13         // bytes necessarios para armazenar o merge de dois contents
14         ci.merge_size=ci.private_size;
15         // bytes necessarios para armazenar a informacao resultante
16         ci.result_size=sizeof(double);
17         // a informacao resultante e valor inteiro ou double?
18         ci.result_is_int=0;
19
20         // funcoes para manipulacao do content
21         static ContentFunctions cfs;
22         ci.pcfs = &cfs;
23         // funcao para capturar parametros adicionais
24         cfs.parse_param=&parse_fator;
25         // funcao para inserir dados no content(apply)
26         cfs.insert=&avg_insert;
27         // funcao para remover dados no content(unapply)
28         cfs.remove=&avg_remove;
29         // funcao para realizar o merge de contents(merge)
30         cfs.merge=&avg_merge;
31         // funcao para extrair informacao do content(extract)
32         cfs.result=&avg_result;
33         // registra o content 'avg' na tecnologia
34         register_content("avg",&ci);
35 }
```

7.4 Módulos Container

Módulos Container se comportam como Contents especiais que, através de um mecanismo de indexação, dão acesso a outros Contents. O valor do índice utilizado nessa indexação é obtido usando os mesmos recursos oferecidos por um Módulo Dimensional. Por conta dessa dupla capacidade, um Módulo Container é instalado simultaneamente como um Módulo Content e como um Módulo Dimensional.

A Listagem 7.5 exibe um código na linguagem C para instalação do Módulo Container "binlist", destacando a segmentação da instalação deste módulo como um Módulo Dimensional e um Módulo Container.

Listagem 7.5: Registro do Conteiner binlist

```
1  void register_content_class_binlist(void) {
2          static Class c;
3          memset(&c,0,sizeof(c)); // Inicializa
4
5          register_class_binlist(&c);    // Registro como Modulo Dimensional
6          register_content_binlist(&c); // Registro como Modulo Content
7  }
```

As listagens 7.6 e 7.7 referem-se, respectivamente, a instalação do Módulo Container como Módulo Dimensional e a instalação das operações definidas para ele, de forma similar ao que foi realizado anteriormente para Módulo Dimensional "geo". Nota-se na linha 3 da Listagem 7.6 que o módulo é definido como CONTAINER_CLASS. Observe que, diferentemente do Módulo Dimensional "geo", este módulo gera chaves para "group-by" com apenas um valor (linha 5).

Listagem 7.6: Registro de binlist como Modulo Dimensional

```
1  static void register_class_binlist(PClass pclass) {
2          pclass->name = "binlist";
3          pclass->class_type = CONTAINER_CLASS;
4
5          pclass->n_group_by_indexes = 1;
6          pclass->group_by_requires_rule = 1;
7
8          pclass->get_distinct_info = &get_distinct_info;
9          pclass->create_op_data = &create_op_data;
10         pclass->destroy_op_data = &destroy_op_data;
11
12         register_class(pclass);
13
14         // Registro dos operadores
15         register_ops(pclass->name);
16 }
```

Listagem 7.7: Registro de binlist como Modulo Content

```
static void register_content_binlist(PClass pclass){
        static ContentFunctions cfs;
        cfs.parse_param = _parse_param;
        cfs.prepare = _prepare;
        cfs.shallow_copy = _shallow_copy;
        cfs.insert = &binlist_insert;
        cfs.remove = &binlist_remove;
        cfs.find_bounds = &binlist_find_bounds;

        static ContentInfo ci;
        memset(&ci,0,sizeof(ContentInfo));

        ci.n_param_fields = 1;ci.min_params = 1;ci.max_params = 1;
        ci.params_size = sizeof(BParams);
        ci.private_size = sizeof(BinList);
        ci.pclass = pclass;
        ci.pcfs = &cfs;
        register_content(pclass->name,&ci);
}
```

Listagem 7.8: Registro da operação between

```
static void register_ops(char * classname) {
        static QOpInfo qoi;

        // informa as funcoes que implementam a operacao
        qoi.op = op_between;// para where
        qoi.op_group_by = op_between_by;// para group-bys

        // descreve os argumentos do operador
        qoi.min_args = 2;qoi.max_args = 2;qoi.fmt = "II";

        // registra a funcao como membro da classe
        register_op(classname, "between", &qoi);
}
```

7.5 Módulos Estruturais

Módulos Estruturais são componentes de baixo nível, responsáveis por implementar a camada abstrata para manipulação das arestas e dos vértices de um tinycube. Assim, os vértices e as arestas são utilizados como elementos abstratos pelos algoritmos de Inserção e de Remoção, podendo ser efetivamente armazenados na memória numa forma diferente da usual, na qual um vértice é alocado dinamicamente num registro e uma aresta utiliza o endereço físico deste registro para acessá-la.

A peculiaridade da tecnologia Tinycubes, com relação à módulos estruturais, é que

cada dimensão pode utilizar uma versão diferente de um módulo estrutural, incluindo a capacidade de que os vértices terminais poderem ter versões diferentes das funções de acesso aos elementos. O objetivo desta flexibilização é permitir que níveis de um Tinycube, que possuam uma quantidade limitada e previamente conhecida de valores distintos, possam ser implementados de uma forma mais compacta, potencialmente promovendo economia de memória, e tornando a implementação mais eficiente. Um exemplo típico seriam os níveis de uma quadtree, nos quais só existem 4 possíveis valores distintos que podem ser armazenados.

Como são estruturas de baixo nível não convém examiná-las em detalhadas no corpo principal deste trabalho, porém, a título de informação, estão relacionadas às funções abstratas que as instâncias de um módulo estrutural devem implementar. Dado que vértices regulares possuem arestas e vértices terminais não, algumas funções são definidas exclusivamente para vértices regulares.

Funções utilizadas por todos os vértices

Função	**Descrição**
create_node	cria um vértice
delete_node	remove um vértice
set_subcube_info	configura detalhes sobre a aresta CUBE
get_subcube_info	recupera detalhes sobre a aresta CUBE
shallow_copy	realiza uma operação de "shallow_copy"

Funções utilizadas apenas por vértices regulares

Função	**Descrição**
locate_child	localiza uma aresta e um vértice em arestas CHILD
get_child	recupera um vértice à partir de uma aresta CHILD
add_child_edge	adiciona uma aresta CHILD a um vértice
remove_child_edge	remove uma aresta CHILD de um vértice
replace_child_on_branch	converte aresta SHARED CHILD em PROPER CHILD
has_single_child	verifica se um vértice so tem uma aresta CHILD
get_children_class	descobre quantas arestas CHILD um vértice possui

Capítulo 8

Tinycubes - Tecnologia

Neste capítulo são apresentados elementos da tecnologia Tinycubes que não fazem parte nem da estrutura de dados Tinycubes e nem do sistema de modularização.

8.1 Arquitetura Cliente x Servidor

A tecnologia Tinycubes utiliza um modelo client-server para atender consultas ad hoc. A tecnologia disponibiliza um servidor que continuamente espera receber consultas através de uma porta de comunicação TCP/IP (porta 8001 *default*). Quando uma consulta é recebida, o servidor a analisa e, se não estiver de acordo com a sintaxe definida na Seção 8.3.1, retorna uma resposta sinalizando o problema. Caso a consulta tenha sido corretamente formulada, o servidor inicia o seu processamento. Ao final do processamento, o servidor codifica as informações retornadas pelo processador de consultas segundo o formato definido na Seção 8.3.3

8.2 Especificação do Schema

Conforme descrito na secção 6.1.1, existem diversos parâmetros que determinam como os objetos tinycubes devem ser criados. Esta seção apresenta como os parâmetros que compõem um Schema são fornecidos para a implementação do servidor Tinycubes.

```
1 "record": {
2   "fields": [
3     { "id":"seconds", "type": "int" },
4     { "id":"lat", "type": "double" },
5     { "id":"lon", "type": "double" },
6     { "id":"proto", "type": "int" },
7     { "id":"sport", "type": "int" },
```

```
8       { "id":"dport", "type": "int" },
9       { "id":"ipackets", "type": "int" },
10      { "id":"ibytes", "type": "int" }
11    ]
12  },
13  "dimensions": [
14    { "id": "location", "length": 25, "class": [ "geo", "lat", "lon" ] },
15    { "id": "proto", "class": ["cat", "proto"]}
16    ],
17  "terminal": {
18    "contents": [
19      { "id": "sum", "formula": [ "sum", "ibytes"] } ,
20      { "id": "hours",  "container": ["binlist", "seconds", "60"],
21        "contents": [
22          {"id": "hsum", "formula":["sum", "ibytes"] },
23          {"id": "havg", "formula":["avg", "ibytes"] }
24        ]
25      }
26    ]
27  },
```

Listagem 8.1: Fragmento de um arquivo schema

Ao iniciar, o servidor Tinycubes lê um arquivo codificado no padrão JSON (default: 'schema.json') que enumera os parâmetros necessários para criação dinâmica dos objetos tinycubes. Existem três elementos essenciais que devem estar presentes neste arquivo: (1) a especificação do Record (Seção 6.2.1), fornecido através do objeto "record", (2) a especificação das dimensões que definirão a estrutura do índice, através do *array* "dimensions" e, (3) a especificação do conteúdo dos terminais, através do objeto terminal. Todos elementos que compõem o arquivo de schema estão listados a seguir:

objeto 'record'

Contém o array 'fields' onde cada elemento é um objeto 'field.record'.

objeto 'field' em 'record'

Define um campo da estrutura *Record*, onde 'id' tem o nome do campo e 'type', um valor dentre { 'int', 'double' }, define o seu tipo.

array 'dimensions'

Contém objetos do tipo 'dimension' que descrevem cada uma das dimensão do Tinycubes. O número de elementos nesse array determina o número de dimensãoes do Tinycubes (valor de $\mathcal{D}$ no *Schema*, ver Seção 6.1.1).

objeto 'dimension' em 'dimensions'

Determina determinando como os valores das arestas da dimensão são obtidos. O campo

'id' neste objeto fornece o nome pelo qual a dimensão será referenciada posteriormente. Como esperado, a dimensão pode conter apenas valor de aresta (dimensão simples) ou pode ser formada por múltiplos valores de aresta que, em conjunto, formam um caminho dimensional simples (dimensão complexa).

Se o valor do campo 'height' (altura) estiver ausente ou for igual a 1, então a dimensão é simples. Neste caso, este objeto também é utilizado para especificar a única aresta da dimensão, sendo que o nome da aresta é o mesmo nome da dimensão. Os demais parâmetros que especificam a aresta podem ser encontrados no objeto 'value' a seguir.

Se o valor do campo 'height' é superior a 1, então a dimensão é complexa. Neste caso, existem duas formas de se especificar a origem dos valores de cada uma das arestas que compõem a dimensão. A primeira forma consiste na especificação do array values (definido a seguir). Já a segunda forma consiste na utilização de um construtor oferecido por uma "classe dimensional" (campo 'class'). Esse construtor é uma "função" que, quando acionada, identifica a altura da dimensão 'height', consulta campos do 'record' e eventuais parâmetros, e cria tantos valores quanto definidos pela altura. Na Listagem 8.1, vê-se que a dimensão com id 'location' é definida utilizando a classe dimensional 'geo', que utiliza os campos 'lat' e 'lon' para fornecer, durante a execução, os 25 valores utilizados pelas 25 arestas da dimensão.

array 'values'

Contém objetos do tipo 'value' que informam como obter o valor de cada aresta de uma dimensão.

objeto 'value'

Contém o campo 'id' que identifica a aresta para uso posterior. O valor da aresta é determinado pelo construtor da "classe simples" especificado no campo 'class'. Este construtor é a implementação de uma função mapeadora (ver 6.2.2), que, quando acionada consulta campos do 'record', eventuais parâmetros e retorna um valor mapeado para ser armazenado nas arestas. Na Listagem 8.1, vê-se que o objeto 'dimension' com id 'proto' utiliza a classe simples 'cat' para especificar o valor da aresta.

objeto 'terminal'

Contém o campo 'contents' que define um array de todos o objetos Content armazenados no terminal.

array 'contents'

Array que define vários objetos Content. Pode ser utilizado no objeto 'terminal' ou como campo 'contents' de um objeto 'content' definido em um objeto 'terminal'.

objeto 'content'

Sempre contém um campo 'id', que é utilizado nas consultas para que possa ser selecionado. Pode ser um Content convencional ou um Container. Se for um Content convencional contém um campo 'fórmula', um array cujo primeiro item é um construtor que indica o nome do Módulo Content que deve ser utilizado e os demais itens são parâmetros para este construtor. Se existe um campo 'container', então este 'content' é um Container. Nesse caso, o objeto 'content', deve possuir um campo 'contents' contendo um array 'contents'. Quando o 'content' é um Container, o valor do campo 'container' é um construtor, idêntico ao utilizado em uma fórmula, porém o primeiro argumento identifica o nome do Módulo Container registrado que deve ser utilizado.

8.3 Linguagem de consulta

Todas as interações entre o servidor e a interface de visualização são realizadas através de um protocolo de comunicação que utiliza strings no padrão JSON, formatadas de acordo com um um conjunto de regras sintáticas que definem uma linguagem para consulta ao servidor e para obtenção de respostas. A definição deste protocolo de comunicação atende à atividade de pesquisa *AP8*.

Essa seção descreve tanto o padrão que deve ser utilizado para confecção de consultas a serem submetidas à tecnologia quanto o padrão esperado nas respostas produzidas por ela. Os detalhes de como este padrão de conversação é efetivamente utilizado pela tecnologia serão apresentados ao se descrever o processador de consultas (Seção 8.4), na qual diversos exemplos serão fornecidos.

8.3.1 Consulta

A linguagem utilizada para solicitação de consultas é baseada em JSON e inspirada em comandos SQL tradicionais.

Listagem 8.2: Exemplo de consulta

```
1 {
2     "id": 296,
3     "select": ["hc"],
4     "group-by": "proto", "group-by-output": "kv",
5     "where": [
6         ["location", "zpoly", 8,
7             -20.11783963049162,-46.43920898437501,
8             -18.843913201134132,-41.52832031250001,
```

```
9            -20.519644202728962,-38.594970703125,
10           -22.228090416784486,-40.42968750000001],
11        ["hours", "between", 1565101800, 1565101849]
12    ]
13 }
```

A Listagem 8.2 ilustra uma consulta na qual solicita-se que todas as informações referentes ao Content "hc" (contagem de eventos) associadas a locais dentro do polígono definido pelas coordenadas especificadas no item "location" e que ocorreram no período de tempo definido no item "hours", sejam selecionadas e agrupadas pelo protocolo do pacote. O resultado do agrupamento deve ser organizado no formato "kv", ou seja, em pares contendo uma chave (key) e o valor(valor).

```
request ::= '{' [id ',' ] statement '}';
id ::= "id" ':' NUMBER;
statement ::= schema | bounds | sql_like_statement;
schema ::= "schema";
bounds ::= "bounds" ':' name ',' where;
sql_like_statement ::= select_where [groupby];
select_where ::= select [where] | where;
select ::= "select" ':' '[' names ']';
where ::= "where" ':' '[' conditions ']';
conditions ::= condition { ',' condition };
condition ::= '[' field ',' operator ',' values ']';
field ::= name;
operator ::= name;
values ::= value { ',' value }
value ::= name | NUMBER
group-by ::= "group-by" ':' '[' fields ']';
names ::= name { ',' name }
name ::= "IDENT"
```

Listagem 8.3: BNF da consulta

8.3.2 Comandos

A seguir estão descritos os comandos aceitos pela linguagem.

schema

Requisita o arquivo json utilizado para configurar a instância do tinycubes em execução.

* Formato: **schema**

bounds

Requisita os valores que delimitam uma dimensão.

* Formato: **bounds** *dimensão* **where-clause**

select

Requisita valores de uma dimensão ou de um Content.

* Formato:

select *lista-de-dimensões* ou *lista-de-contents* **[where]**

where

Enumera critérios para seleção das arestas a serem utilizadas na construção das respostas.

* Formato: **where** *lista-de-critérios*

group-by

Agrega as informações selecionadas pela cláusula where em torno dos item expecificado.

* Formato: **group-by** *dimensão* ou *content*

8.3.3 Resposta

O formato das respostas para às consultas será descrito nesta seção. A Listagem 8.4 apresenta o BNF que descreve o formato das respostas produzidas pelo servidor Tinycubes. Todas as respostas possuem um campo "ms" que indica o tempo de execução da consulta e um "tp", que indica o tipo de resultado que o campo "result" contém. Se o valor de "tp" for: 0-houve um erro na consulta, 1-result tem todo o conteúdo do arquivo JSON que descreve o "schema", 2-"info" está no formado "kvs", 3-"info" está no formato "vs" e 4-"info" está no formato "full".

Listagem 8.4: BNF da resposta

```
response ::= '{' [id ',' ] tp ',' result ',' ms '}';
id ::= "id" ':' NUMBER;
tp ::= 'tp' = NUMBER;
ms ::= "ms" ':' NUMBER;
result ::= "result" ':' info;
info ::= kvs | full | vs | schema;
kvs ::= '[' [ kv { ',' kv } ] ']
kv ::= '{' "k" ':' '[' numbers ']' ',' "v" ':' '[' numbers ']' '}'
vs ::= "vs" ':' '[' numbers ']'
full ::= '{' "ks" ':' '[' numbers ']', "vs" ':' '[' numbers ']' '}'
numbers :: = NUMBER { ', ' NUMBER }
```

8.4 Processador de consultas agnóstico - AQP

Esta seção oferece uma visão geral do processamento de consultas implementado no servidor Tinycubes que realiza a atividade de pesquisa *AP9*. Uma consulta ad-hoc é uma consulta que, embora deva obedecer a algumas regras de formação, não está limitada a um formato demasiadamente rígido, como ocorre com a especificação dos parâmetros que uma função aceita.

O objetivo principal do processador de consultas da tecnologia Tinycubes, doravante também chamado de AQP (Agnostic Query Processor), é recuperar as informações encontradas à partir de especificações que definem um subconjunto dos dados recebidos a ser utilizado nesta busca. Por exemplo, num cenário onde a tecnologia recebe dados contendo uma coordenada geográfica e um valor, é possível construir um Schema no qual consultas podem demandar a soma dos valores recebidos (informação) à partir da definição de uma área geográfica (dados recebidos).

A ideia básica do processador de consultas é navegar pelas as arestas de um tinycube, seguindo as orientações fornecidas pela consulta e obter a informação desejada no vértice terminal encontrado. No entanto, é necessário destacar alguns pontos. Em primeiro lugar, deve-se lembrar que uma árvore de busca define diversas partições. A cada escolha de aresta a ser percorrida, está se definindo uma partição dos possíveis Contents a serem encontrados. Além disso, como um vértice terminal pode armazenar diversos Contents, é necessário especificar qual Content será utilizado para gerar a informação.

Existem situações que o subconjunto de busca desejado é formado pela união de diversas partições. Neste caso, uma mesma consulta deve visitar vários vértices terminais, coletando o resultado. A partir daí existem duas possibilidades. Na primeira, o processador agrega (utilizando a função *merge*) o valor dos Contents encontrados e retorna uma única informação. Essa agregação sempre se inicia criando um acumulador de resultado (AR), inicializado com o Content C_0, conforme definido na Seção 6.2.3.

O segundo caso é muito importante e é a base para a definição da operação "group-by". Neste caso, a consulta orienta o processador a agrupar os Contents encontrados de acordo com uma parte dos valores utilizados para localizá-los. Operações de "group-by" são as operações tipicamente utilizadas para criar histogramas. Como exemplo do funcionamento dessa operação, considere uma tabela com as colunas "estado", "cidade", "número de habitantes". Uma operação de "group-by" utilizando como localizador a coluna "estado" e como valor resultante o "número de habitantes" produziria, como resultado

final, uma relação com duas colunas, uma delas contendo a informação do "estado" e na outra a agregação (total) do "número de habitantes" por "estado". Resumindo, uma operação "group-by" cria grupos de informações identificados por chaves, compostas pelo(s) valor(es) utilizado(s) para localizar as informações do resultado, e atribui para a cada grupo, o Content combinado ("merge") de todas informações do resultado que foram localizadas utilizando a chave que identifica o grupo.

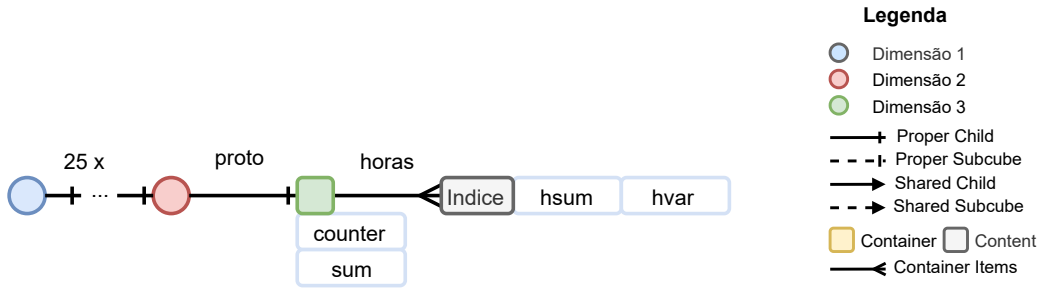


Figura 8.1: Schema descrito na Listagem 8.1

As seções a seguir descrevem o funcionamento do processador de consulta nas duas situações através de exemplos com complexidade crescente. Primeiro descrevendo o processamento de consultas que não usam o "group-by" por serem consultas mais simples, sendo, por esta razão, denominadas consultas simples. Em seguida, descrevem as consultas que utilizam "group-by".

Todos os exemplos utilizam o Schema definido na Listagem 8.1 e ilustrado na Figura 8.1. Nesta figura, vêm-se as três dimensões do esquema listadas na posição horizontal. Como novidade, existe um retângulo com cor de fundo amarela identificado por "índice" e retângulos adicionais com cor de fundo "branca" identificados por outros nomes. O retângulo com cor amarela simboliza uma das possíveis instâncias dos item que um Container pode ter. Os retângulos a sua direita, indicam os Contents que cada um desses itens armazena. No caso do Schema da Listagem 8.1, cada item armazena dois Contents, identificados pode "hsum" e "hvar". Já os retângulos com fundo branco dispostos abaixo do vértice terminal, identificados por "counter" e "sum", representam os Contents armazenados diretamente no vértice terminal. Enquanto o Content "sum" foi explicitamente definido na listagem do schema, o Content "counter" é criado automaticamente pela tecnologia e armazena a diferença entre a quantidade de inserções e de remoções no vértice terminal.

8.4.1 Consultas sem "group-by" (consulta simples)

Uma consulta simples consiste na localização de um vértice terminal seguida da extração da informação do Content desejado. Eventualmente, uma consulta simples pode exigir a agregação dos Contents de vários vértices para se obter a informação desejada. Nesse caso, o AQP irá inicializar um acumulador de resultado (AR) que é um Content inicializado com o valor C_0 (ver Seção 6.2.3) e, para cada vértice localizado, extrair do vértice o Content especificado aplicando, em seguida, a função *merge* sobre o AR e sobre o valor do Content desejado, armazenando o resultado no AR. Uma vez que não exista mais vértices terminais a visitar, extrai-se a informação do Content no AR. O processo para localizar o(s) vértice(s) terminal(is) adequado(s) depende da estrutura da consulta solicitada e das operações de seleção utilizadas pela consulta.

A Listagem 8.5 exibe um exemplo de consulta que solicita o valor do Content "sum". Como não existe nenhuma restrição referente ao conjunto de dados a ser considerado, o processador "entende" que deseja-se o total de "sum", ou seja, à partir do Maxiroot, deve-se seguir por arestas CUBE até chegar ao vértice terminal que armazena o Content correspondente a agregação de todos os demais vértices terminas da Maxitree e retornar a informação extraída dele.

Listagem 8.5: Consulta elementar

```
1  {
2    "id": 10,
3    "select": ["sum"],
4  }
```

Listagem 8.6: Consulta básica

```
1  {
2    "id": 11,
3    "select": ["sum"],
4    "where": [  ["proto", "eq", 6] ]
5  }
```

A Listagem 8.6 exibe uma consulta simples que apresenta uma restrição sobre o conjunto de dados a ser considerado, refente ao id "proto" do Schema. Nesse caso, a consulta expressa o desejo de que se obtenha o valor "sum" apenas para o protocolo com valor igual[1] a "6" (código para o protocolo TCP). Diante desta consulta, o AQP (processador de consultas), sempre iniciando do Maxiroot, utiliza a aresta CUBE para pular toda a dimensão 1 ("location") e atingir um vértice "v", que é um vértice TOP da dimensão 2 ("proto"). Uma vez encontrado o vértice TOP correto da dimensão 2, o AQP, passa a chamar a rotina correspondente à operação "eq" para cada uma das arestas CHILD filhas de v, até que a rotina retorne TRUE para alguma aresta a. Caso a rotina nunca retorne TRUE, o

[1] assumindo que "eq" seja uma operação que retorna verdadeiro se o valor armazenado na dimensão identificada por "proto" seja igual ao valor do argumento

processador de consultas simplesmente retorna e a resposta não informará nenhum valor. Caso a rotina retorne TRUE, o AQP chama recursivamente[2] a si mesmo, seguindo para o vértice apontado por essa aresta a e passa a seguir arestas CUBE até atingir o vértice terminal, que é o vértice desejado. A partir daí o AQP extrai as informações do Content relacionado a "sum" e a consulta está terminada.

Listagem 8.7: Exemplo de consulta com agregação de Contents

```
1 {
2   "id": 12,
3   "select": ["sum"],
4   "where": [  ["proto", "eq", 6, 17] ]
5 }
```

A Listagem 8.7 exibe uma consulta simples que exige a aplicação da função *merge*. Como se pode perceber, essa consulta é quase idêntica a anterior, com à exceção de que existem dois números após o operador "eq". Para o AQP, isso indica que a rotina "eq" receberá uma lista de números a serem avaliados. O processo é idêntico ao ao anterior, o AQP chama a rotina "eq" para cada valor de aresta encontrado. No entanto, na consulta anterior, devido às propriedades do Tinycubes, apenas uma aresta era encontrada e para esta consulta podem ser até duas. O que ocorre, é que, o AQP chama a si mesmo recursivamente, para percorrer todas as arestas autorizadas pelas funções operadoras, porém utilizando um único AR. A cada vez que o AQP atinge um vértice terminal, ele faz um "merge" com o AR utilizado. No exemplo, se um vértice tiver arestas com valor "6" e "17", então os Contents presentes nos vértices terminais alcançados através dessas arestas seriam combinados ("merged"). No mundo real, a informação resultante seria a soma total de todos os bytes recebidos para os protocolos TCP e UDP.

Listagem 8.8: Exemplo de consulta com restrição de tempo

```
1 {
2   "id": 13,
3   "select": ["sum"],
4   "where": [
5     ["proto", "eq", 6],
6     ["hours", "between", 1565101800, 1565101849]
7   ]
8 }
```

A Listagem 8.8 exibe uma consulta simples que exige que duas operações retornem TRUE e ilustra uma das formas de como um Container "hours" (ver Figura 8.1) é pro-

[2]Essa chamada recursiva é uma forma de preservar o ponto da busca por arestas CHILD com o valor especificado, antes de continuar a descer pela estrutura, visando uma possível retomada do processo de inspeção das arestas filhas de v

cessado em uma consulta simples. Neste caso, a maior parte da consulta se dá como no exemplo baseado na Listagem 8.6, exceto que o AQP percebe que só pode retornar o Content "sum" de um vértice terminal v, se a função que implementa o operador "between", retornar TRUE quando chamada tendo este vértice v como parâmetro. Cabe ao responsável por implementar a função "between", definir em que condições ela retornará TRUE. No exemplo, isso ocorre quando existir pelo menos um Content armazenado com um valor de índice[3] entre os valores das duas constantes passadas como parâmetro[4]. Em resumo, a informação retornada é extraída do Content "sum" armazenado no vértice terminal e não da agregação (*merge*) do Content "hsum" de todos os elementos no Container "hours". E essa informação só é retornada se existir qualquer Content armazenado no Container dentro do intervalo especificado. Caso contrário a resposta à consulta não conterá informações.

Listagem 8.9: Exemplo de consulta temporal

```
1  {
2    "id": 14,
3    "select": ["hsum"],
4    "where": [
5      ["proto", "eq", 6],
6      ["hours", "between", 1565101800, 1565101849]
7    ]
8  }
```

A Listagem 8.9 exibe uma versão da consulta anterior, na qual espera-se a soma (agregação) de todos os valores armazenados nos Contents "hsum" do Container "hours" que estejam dentro período selecionado. A diferença entre esta e a consulta anterior é o Content selecionado. Na consulta anterior, selecionou-se o Content "sum" (Content do terminal) e nessa usa-se o Content "hsum" (Content do Container).

Listagem 8.10: Exemplo de consulta espacial

```
1  {
2    "id": 15,
3    "select": ["sum"],
4    "where": [
5      ["location", "zpoly", 8,
6        -20.11783963049162,-46.43920898437501,
7        -18.843913201134132,-41.52832031250001,
8        -20.519644202728962,-38.594970703125,
9        -22.228090416784486,-40.42968750000001]
10     ],
11     ["proto", "eq", 6]
12   }
```

[3]ver definição de Container na Seção 6.2.3

[4]essas constantes representam o tempo em segundos contados à partir de uma data base.

A Listagem 8.10 exibe uma consulta envolvendo uma dimensão inteira, por utilizar a classe dimensional "location". A consulta utiliza, além do operador "eq", o operador "zpoly" para selecionar quais vértices terminais devem fornecer Contents para composição da informação resultante. O operador "zpoly" consulta o nível de zoom especificado no primeiro parâmetro (no exemplo, o valor 8) e o polígono formado pela sequência de pontos geográficos (latitude, longitude) à partir do segundo parâmetro para decidir quais arestas da dimensão "location" de cada altura dimensional devem ou não ser utilizadas para encontrar um vértice terminal. Enquanto os polígonos definem a área geográfica de interesse, o nível de zoom informa para a função "zpoly" em que altura dimensional não é mais necessário seguir arestas CHILD e pode-se utilizar a aresta CUBE para pular diretamente para o vértice TOP da próxima dimensão. A partir daí o restante da consulta se comporta como na Listagem 8.6.

8.4.2 Consultas com "group-by"

Listagem 8.11: Exemplo de group-by básico

```
1 {
2   "id": 20,
3   "select": ["sum"],
4   "group-by": "proto"
5 }
```

A Listagem 8.11 exibe uma consulta básica utilizando o "group-by" que produz, como resultado, a soma total de todos os bytes recebidos, agrupados por protocolo. Para realizar essa consulta, o AQP inicialmente tem de identificar a classe "C" do "elemento" que agrupará os resultados. No caso, o elemento é "proto" e a a classe é "cat". Com base nessas informações, o AQP chama o método "get_distinct_info" dessa classe "C", e descobre qual a quantidade N de elementos distintos possíveis, dentre outras informações. Depois, aloca e inicializa a memória suficiente para armazenar N acumuladores de resultado do tipo de Content solicitado na consulta (no caso do exemplo, "sum"). Em seguida, o AQP inicia a consulta quase como se fosse uma consulta simples, porém com duas diferenças. A primeira é que o vértice correspondente ao nível do elemento no "group-by" sempre é visitado e todas as arestas CHILD encontradas são percorridas recursivamente (como se houvesse um "eq" com o valor de cada uma delas). A segunda é que, ao atingir um vértice terminal, o Content é combinado ("merged") com o AR armazenado na posição correspondente ao valor OPTION encontrado na aresta CHILD que levou até o vértice terminal.

Listagem 8.12: Exemplo de group-by com agregação de Contents

```
{
  "id": 21,
  "select": ["sum"],
  "where": [ ["proto", "eq", 6, 17] ]
  "group-by": "proto"
}
```

A Listagem 8.12 exibe uma outra consulta utilizando "group-by". A diferença nesta consulta é a cláusula "where" aplicada sobre o mesmo "elemento" utilizado pelo "group-by". Neste caso, o AQP procede como na consulta na Listagem 8.11, com uma diferença. Ao invés de aceitar todas as arestas CHILD no nível de "proto", o AQP só aceita as arestas para as quais o operador "eq" retornou TRUE. Assim, o resultado desta consulta é contém apenas as somas totais para os protocolos TCP e UDP.

Listagem 8.13: Exemplo de group-by sobre Container

```
{
  "id": 22,
  "select": ["hsum"],
  "where": [
    ["proto", "eq", 6 ],
    ["hours", "between", 1565101800, 1565101849]
   ]
  "group-by": "hours"
}
```

A Listagem 8.13 exibe uma consulta onde o "group-by" é realizado sobre o Container "hour". Neste caso, o AQP criará um AR para cada minuto do intervalo[5] e, para cada vértice terminal descendente de uma aresta CHILD com valor "6" no nível "proto" e, para cada valor do índice do Container dentro dos limites definido pelo operador "between", o valor do Content "hsum" sera combinado com o valor do AR correspondente. O resultado final desta consulta é uma tabela com o total acumulado de bytes recebidos para o protocolo TCP para cada minuto do intervalo definido no operador "between".

Listagem 8.14: Exemplo de group-by sobre Container

```
{
  "id": 23,
  "select": ["counter"],
  "where": [
    ["proto", "eq", 6 ],
    ["hours", "between", 1565101800, 1565101849],
    ["location", "zpoly", 8,
      -20.11783963049162,-46.43920898437501,
      -18.843913201134132,-41.52832031250001,
```

[5] o Schema define o tamanho do "bin" como 60 segundos na linha 20 da Listagem 8.1.

```
10        -20.519644202728962,-38.594970703125,
11        -22.228090416784486,-40.42968750000001]
12      ],
13    ]
14    "group-by": "location"
15    }
```

A Listagem 8.14 apresenta uma consulta com três categorias de seletividade: categórica ("proto"), temporal ("hours") e espacial ("location"). O id "counter" utilizado no "select" é um campo criado automaticamente em todo vértice terminal e registra o número de inserções ativas no vértice. O operador "zpoly" está sendo usado novamente, mas como, nesta consulta, a dimensão associada a ele é usada por um "group-by", serão fornecidos mais alguns detalhes do seu funcionamento interno.

A operação "zpoly" foi implementada na classe "geo" criando-se um "frame buffer" virtual, no qual o polígono definido pelas coordenadas fornecidas como parâmetro é "desenhado". O número de posições (resolução) deste "frame buffer" depende do parâmetro zoom da operação. Durante a consulta, a função "zpoly" verifica se a aresta correspondente à coordenada geográfica, deve ou não ser utilizada na produção de informação. Em caso afirmativo, o AQP continua seguindo arestas da estrutura tentando chegar a um vértice terminal. Em caso negativo, o AQP simplesmente retorna da chamada eventualmente recursiva.

Neste tipo de consulta, o AQP, como sempre, inicia o percurso pelo Maxiroot e utiliza o mesmo critério para selecionar as arestas de um "group-by", com uma diferença em relação ao processamento da consulta da Listagem 8.13. O valor que localiza os acumuladores de resultados agora é bidimensional, dado que é uma coordenada no plano, e não mais unidimensional. No restante, a consulta funciona como nos casos anteriores. O resultado da consulta é um "heatmap" como o da Figura 9.20, no qual cada célula (tile), identificada por uma coordenada bidimensional (na resolução definida pelo zoom), contém a quantidade de inserções realizadas na sua área geográfica correspondente.

8.4.3 Processador de Consultas Agnóstico

O processador de consultas utilizado pelo Tinycubes é capaz de realizar consultas simples e consultas envolvendo operações de group-by sem avaliar diretamente qualquer valor de aresta. A interpretação se uma aresta deve ser percorrida ou não durante uma consulta é de responsabilidade de rotinas externas ao processador de consultas, que implementam as operações especificadas nas consultas. Até mesmo elementos dimensionais hierárquicos,

como a dimensão geográfica “location”, são tratados pelo AQP de forma indistinta de outros tipos de dimensão hierárquica. Não existe nenhum privilégio ou especificidade para qualquer tipo de dado. Como para o AQP, os valores das arestas não tem significado e não possuem nenhuma forma de predileção, ele é denominado Processador de Consultas Agnóstico.

Capítulo 9

Resultados e Caso de uso

Este capítulo apresenta os resultados de experimentos realizados com uma implementação da tecnologia Tinycubes e um caso de uso da tecnologia na criação da ferramenta NetworkBorescope.

9.1 Configurações e programas utilizados

Os algoritmos principais e rotinas auxiliares que compõem a tecnologia foram implementados utilizando a linguagem C, conforme o padrão C11. Todo o código fonte foi compilado com o compilador gcc versão 7.4.0. As bibliotecas externas utilizadas foram mongoose (CESANTA), para implementação de um servidor HTTP e a biblioteca cJSON, para leitura de arquivos no formato JSON. Os resultados dos experimentos apresentados nesta seção foram realizados em um computador utilizando um processador para desktop Intel Core-i7 3770 (lançado em Q2/2012) e 16GiB de RAM DDR3 com o sistema operacional Linux Ubuntu 18.04.

Também foi implementada, para fins de comparação, uma versão otimizada do Nanocubes, batizada de Nanocubes+. Nessa versão, a estrutura não utiliza vértices de ligação, porém o algoritmo de inserção tem comportamento praticamente idêntico ao utilizado pelo algoritmo original do Nanocubes. A única diferença decorre da forma de acesso os vértices da estrutura, justamente devido à ausência de vértices de ligação. Como existem menos vértices para alocar e acessar, é esperado que o algoritmo dessa versão otimizada ocupe menos memória e também seja mais rápido do que o algoritmo da versão original.

Deve-se registrar que a implementação utilizada no artigo do Nanocubes utilizou técnicas agressivas para redução do consumo de memória, chegando a guardar informação em

alguns bits de ponteiros de 64 bits e usar uma biblioteca otimizada para alocar memória ("libtcmalloc"). Por outro lado, a implementação desenvolvida neste trabalho não utilizou técnicas tão agressivas. Todas as otimizações usaram facilidades comuns de compiladores C, como campos de bits (bitfields) e um "pragma" que mantém os dados das **structs** com seu tamanho real, evitando alinhamentos forçados a endereços múltiplos de 4 ou 8 byytes.

9.2 Resultados com datasets de validação

Nesta seção são apresentados resultados de experimentos que avaliaram os novos algoritmos de inserção e remoção do Tinycubes quanto a correção, ou seja, se a implementação dos algoritmos produziria os resultados esperados ao ser submetida a conjuntos de dados sintéticos projetados para testar condições sensíveis, incluindo as condições que levavam ao aparecimento de peculiaridades no Nanocubes. Os resultados dos experimentos realizados nesta seção são gráficos produzidos à partir de informações coletadas durante a execução, em particular, após a estrutura executar uma operação de inserção ou de remoção.

9.2.1 Introdução

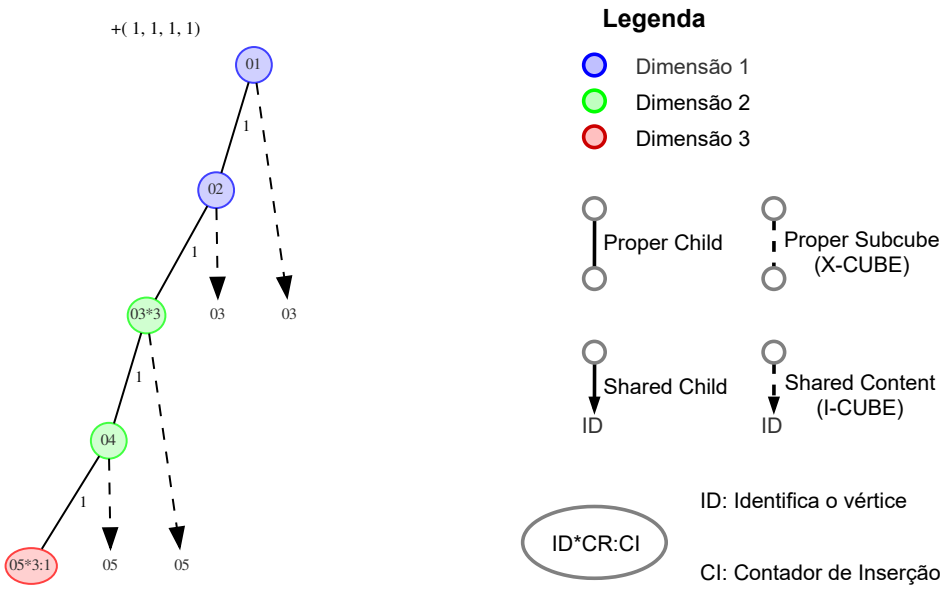


Figura 9.1: Representação visual de tinycube produzida pelo programa "dot"

A confecção dos gráficos nas figuras foi automatizada como forma de evitar que o processo de avaliação pudesse sofrer com problemas introduzidos por ação humana, como

lapsos. O software utilizado para a geração das imagens foi o programa "dot", que pode ser obtido no site "https://graphviz.org". Esse software possui limitações quanto as opções gráficas disponíveis, forçando a mudança no padrão de exibição de tinycubes (e nanocubes) e nas legendas. Além disso, como o programa gera as imagens automaticamente, as arestas do tipo SHARED tiveram de sofrer modificações.

A Figura 9.1 exibe as modificações no padrão de exibição dos Tinycubes. Pode-se ver que todos os vértices possuem um valor, chamado ID, que serve para identificá-lo. Todas as arestas do tipo SHARED, ao invés de terminarem no vértice destino, apontarão para o identificador de vértice (ID). Além disso, todos os vértices compartilhados apresentam apresentam um contador de referências (CR) que indica quantas vezes o vértice está sendo referenciado. Vértices terminais também possuem um contador de inserção (CI), que informa, para fins de acompanhamento do algoritmo, qual a quantidade de inserções que superaram as remoções no vértice. Note também que as arestas PROPER perderam a marca que indicava o seu sentido, obrigando que o vértice destino sempre seja posicionado mais abaixo do que o vértice origem. A figura também indica, na parte superior, qual operação foi executada para deixar o tinycube no estado atual. Finalizando a lista de modificações, os vértices que são subraízes de Tinytrees tem as letras escritas em vermelho.

Devido ao elevado volume de vértices e arestas associados a um Tinycubes, não é possível gerar imagens de estruturas decorrentes de muitas inserções. Desta forma, os testes gerados foram cuidadosamente construídos para formar um conjunto pequeno com operações que pudesse inspecionar se as peculiaridades encontradas na estrutura Nanocubes e no seu o algoritmo de inserção foram resolvidas pela estrutura Tinycubes e seu algoritmo de inserção. A seguir serão apresentados as figuras decorrentes dos testes qualitativos realizados juntamente com comentários adicionais quando necessário. Para realização desses testes utilizou-se $D = 2$ e $\mathcal{H} = [2, 2]$.

9.2.2 Sequência de Inserções - Nanocubes+

A atividade de pesquisa *AP4* tem como objetivo o desenvolvimento de um algoritmo de inserção para o Tinycubes com eficiência superior ao algoritmo que executa a mesma função na estrutura Nanocubes, caso isso seja possível. Nesta seção, serão apresentadas figuras geradas durante uma sequências de inserção de endereços utilizando o programa Nanocubes+ e sua versão otimizada do algoritmo de inserção do Nanocubes. Essas imagens revelam como as peculiaridades do algoritmo original afetam a construção da estrutura. Posteriormente, esses resultados serão utilizados como referência para evidenciar as me-

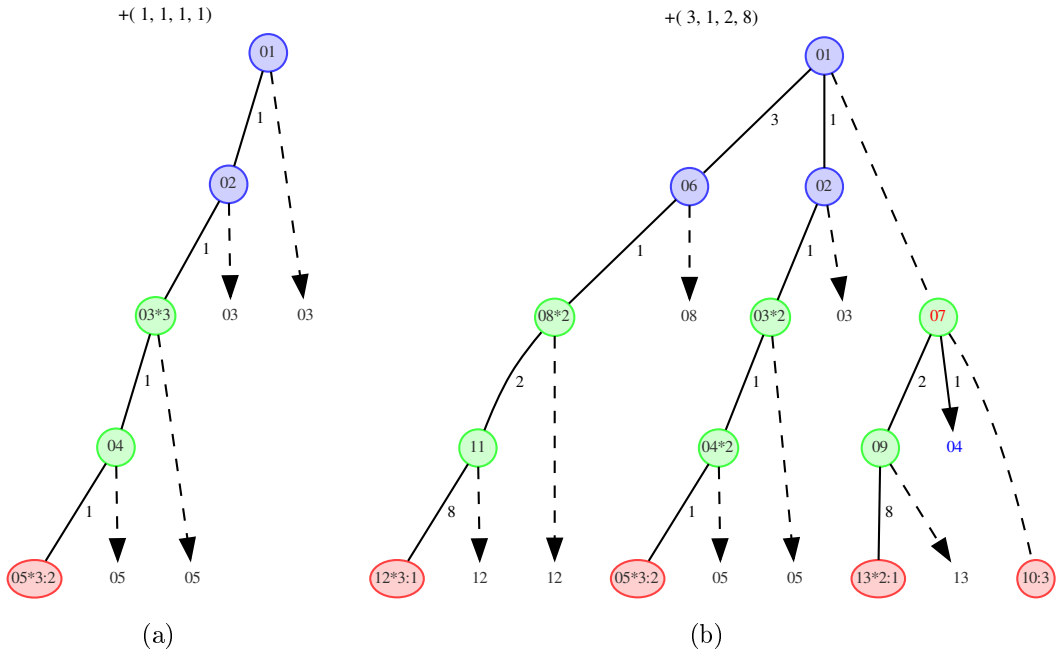


Figura 9.2: Consequências da peculiaridade 1

lhorias do algoritmo de inserção do Tinycubes.

A Figure 9.2(a) exibe o resultado da reinserção do endereço (1, 1, 1, 1) no nanocube+ da Figura 9.1. É possível perceber que essa a reinserção limitou-se a incrementar o CI do vértice sumário[1] indicado pelo endereço e nenhuma situação indesejável ocorreu.

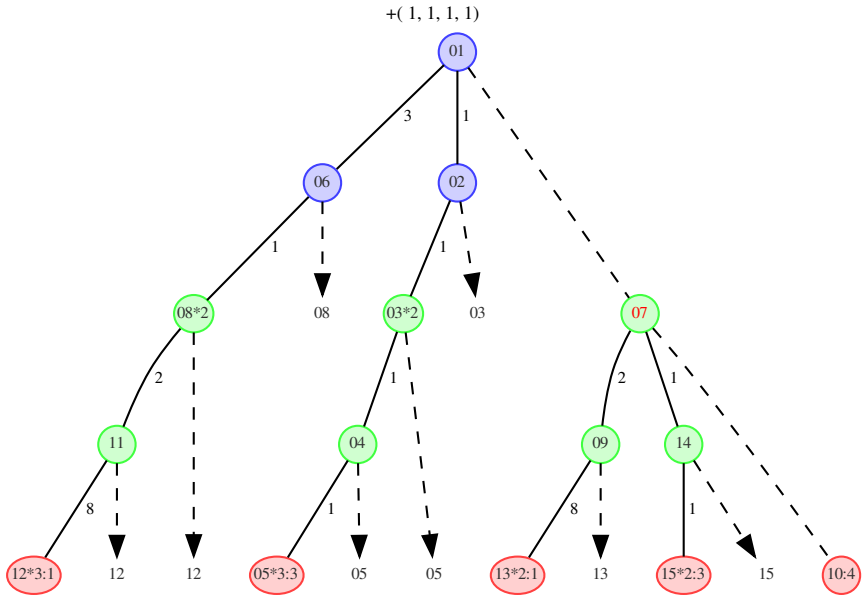


Figura 9.3: Consequências da peculiaridade 2

A Figure 9.2(b) exibe o resultado da inserção do endereço (3, 1, 2, 8). Note que,

[1] vértice terminal no padrão Tinycubes

utilizando apenas arestas CHILD, existem apenas dois caminhos possíveis à partir do vértice superior até o vértice sumário. Isso deveria ser particularmente útil na dimensão 2, uma vez que não existem duas arestas com o mesmo valor logo após o primeiro vértice da segunda dimensão. Assim, teoricamente seria possível que todas as arestas com origem no vértice "07" fossem do tipo SHARED CHILD, aproveitando da ausência de valores iguais neste mesmo nível.

No entanto, não é isso o que ocorre na prática. Devido a peculiaridade 1, identificada na atividade de pesquisa *AP2*, o algoritmo de inserção não aproveita essa oportunidade. Como resultado, é criada uma aresta PROPER CHILD com valor 2 e origem no vértice "07". Note que, à partir da criação desta aresta, é necessário criar todo um novo caminho até um novo vértice sumário, potencializando o custo desta aresta PROPER CHILD indevida. Note ainda que o próprio vértice sumário tem um custo diferenciado, uma vez que ele pode armazenar grandes séries temporais, o que aumenta ainda mais o custo total gerado pela ocorrência dessa peculiaridade 1. E ainda observe que, caso o novo vértice "09" aponte para outra aresta CHILD, será necessário criar um novo vértice sumário, apontado por uma aresta PROPER CONTENT com origem no vértice "09".

A Figure 9.3(a) exibe o resultado a inserção do endereço (1, 1, 1, 1) no nanocube+ da Figura 9.2(b). Esta inserção é, na verdade, mais uma reinserção do endereço (1, 1, 1, 1). Note que, diferentemente do que ocorreu na reinserção ilustrada na Figura 9.2(a), foi necessário ajustar os descendentes do vértice "01" alcançáveis através da aresta CONTENT[2]. Quando o vértice "07" foi encontrado, o algoritmo de inserção apresentou a peculiaridade 2 e começou a criar vértices até criar um novo vértice sumário. Observe que este vértice sumário recém-criado (vértice "15") tem o mesmo valor do vértice sumário que teve o CI atualizado para 3 (vértice "05"), mantendo a consistência da estrutura. Porém, como foi visto na análise da peculiaridade 2, toda este conjuntos de novas arestas e vértices não deveria existir.

A partir do que foi exposto nesta seção, as Figuras 9.2(b) e 9.3 serão utilizadas, respectivamente, como exemplo das peculiaridades 1 e 2, que devem ser evitadas no novo algoritmo de inserção. Os testes subsequentes verificarão se as inserções de endereço que causaram o aparecimento das peculiaridades nas figuras selecionadas, terão sido solucionadas ou não.

[2]aresta CUBE no padrão Tinycubes

9.2.3 Sequência de Inserções

A Figura 9.1 apresenta o resultado da inserção do endereço (1, 1, 1, 1) num tinycube que inicialmente continha apenas o vértice Maxiroot[3]. O resultado está de acordo com o esperado: as arestas SHARED CONTENT tendo sido criadas corretamente e o CI (contador inserções) apresentando o valor 1.

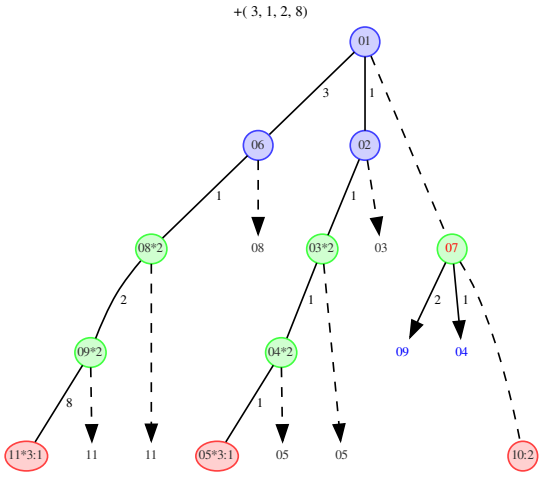


Figura 9.4: Exemplo de superação da peculiaridade 1 do Nanocubes

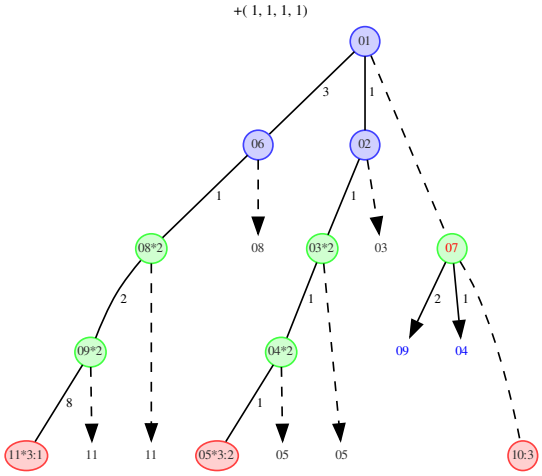


Figura 9.5: Exemplo de superação da peculiaridade 2 do Nanocubes

[3]Como nesta sequência não existe uma dupla inserção do endereço (1,1,1,1), os valores de CI das figuras a seguir e das figuras da sequência de inserções com o Nanocubes+, será ligeiramente diferente

A Figura 9.4 mostra um caso onde o novo algoritmo não apresenta a peculiaridade 1 do algoritmo de inserção do Nanocubes. Na figura, vê-se o resultado da inserção do endereço (3, 1, 2, 8). Por conta da bifurcação no Maxiroot, foi criada uma tinytree com raiz no vértice "07". Comparando esta figura com a Figura 9.2(b), conclui-se que a presença das duas arestas SHARED CHILD com origem no vértice "07" demonstram que a incapacidade de estabelecer compartilhamento de arestas inéditas foi resolvida. Nota-se também que, mesmo sem a criação de arestas PROPER CHILD no vértice "07", o valor do CI do vértice terminal mais a direita (vértice "10") foi corretamente atualizado.

A Figura 9.5 mostra um caso onde o novo algoritmo de inserção não apresenta a peculiaridade 2 do algoritmo de inserção do Nanocubes (reinserção). Na figura, vê-se o resultado da inserção do endereço (1, 1, 1, 1). Note que esta reinserção ajustou os CIs (Contadores de Inserção) dos vértices "05" e "10" sem criar nenhum novo vértice ou nova aresta.

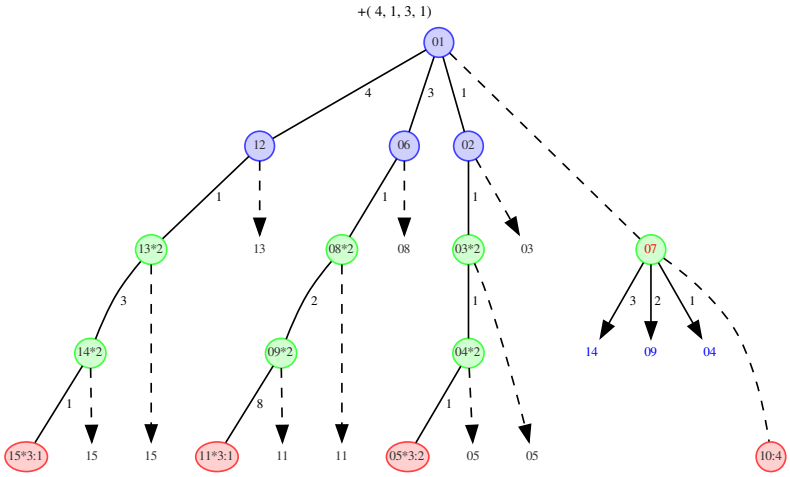


Figura 9.6: Criação de SHARED CHILD para compartilhamento

A Figura 9.6 exibe o resultado da inserção do endereço (4, 1, 3, 1), que provoca a criação de uma nova aresta PROPER CHILD filha do Maxiroot, iniciando um novo caminho até o vértice terminal "15". Com existe um novo valor na primeira altura da dimensão 2, uma aresta SHARED CHILD no vértice "07" é criada apontando para o novo vértice da mesma altura da dimensão 2.

A Figura 9.7 exibe o resultado da inserção do endereço (1, 2, 4, 1). Destacam-se: a criação de uma nova tinytree, cuja raiz é o vértice "17", possuindo duas arestas devido a nova bifurcação, a nova aresta SHARED na tinytree (raiz no vértice "07") com quarto

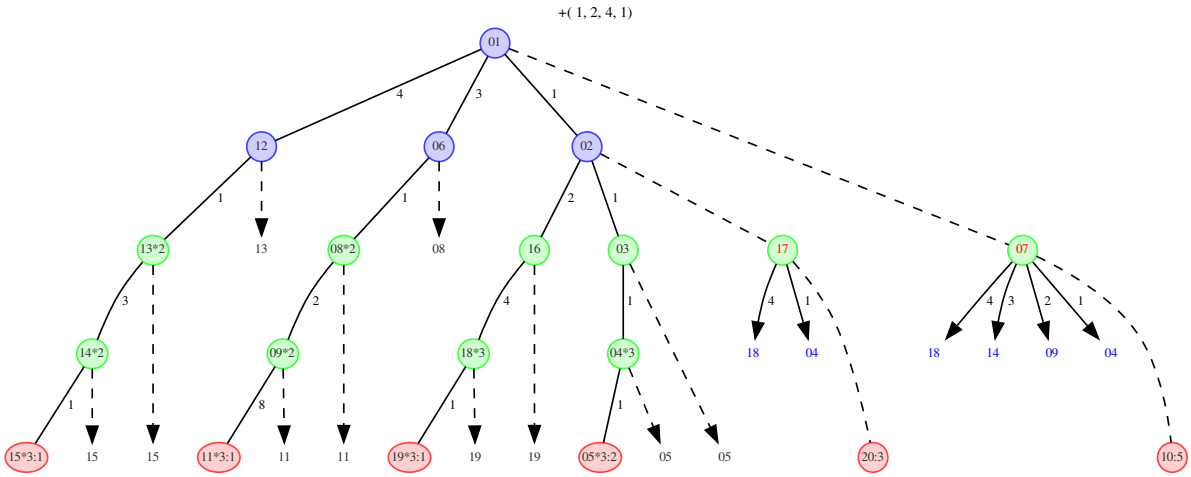


Figura 9.7: Criação de mais uma tinytree sem apresentar a peculiaridade 1

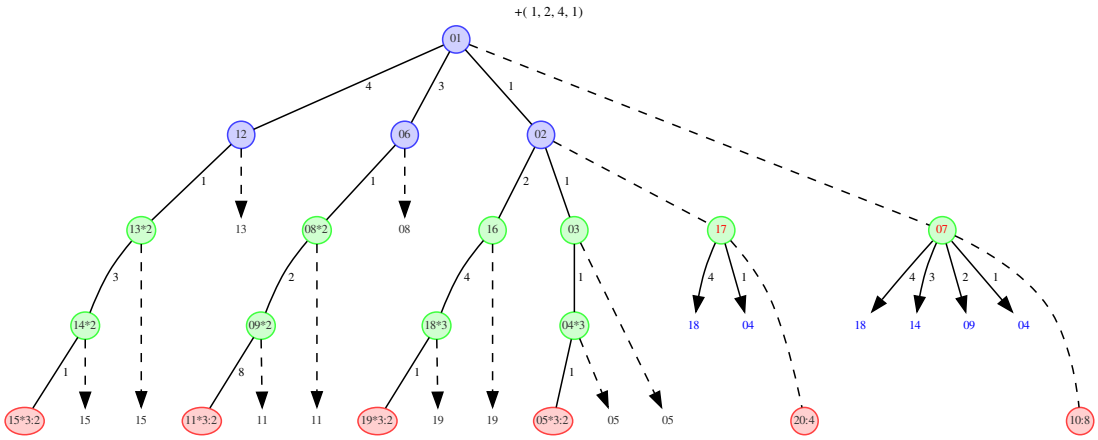


Figura 9.8: Diversas reinserções sem ocorrência da peculiaridade 2

arestas (sem peculiaridade 1) e atualização de todos os CIs.

A Figura 9.8 realiza apenas atualização de CIs, sem apresentar a peculiaridade 2, apesar das reinserções dos endereços (3, 1, 2, 8), (4,1,3,1), (1,2,4,1).

A Figura 9.9 é resultado da bifurcação na dimensão 2, decorrente da inserção do endereço (1,1,2,3), e suas consequências (sem peculiaridade 1). O vértice "07" que possuía 4 arestas SHARED CHILD, necessita criar uma aresta PROPER porque agora existem duas arestas com o valor 2 que são descendentes do seu pai (vértice "01") no mesmo nível dimensional. No entanto, o vértice "21", criado para ser destino dessa nova aresta

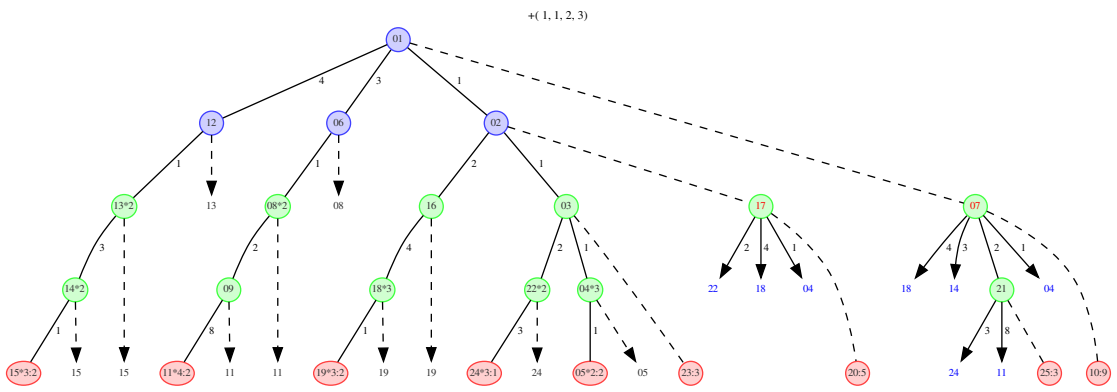


Figura 9.9: Bifurcação na dimensão 2

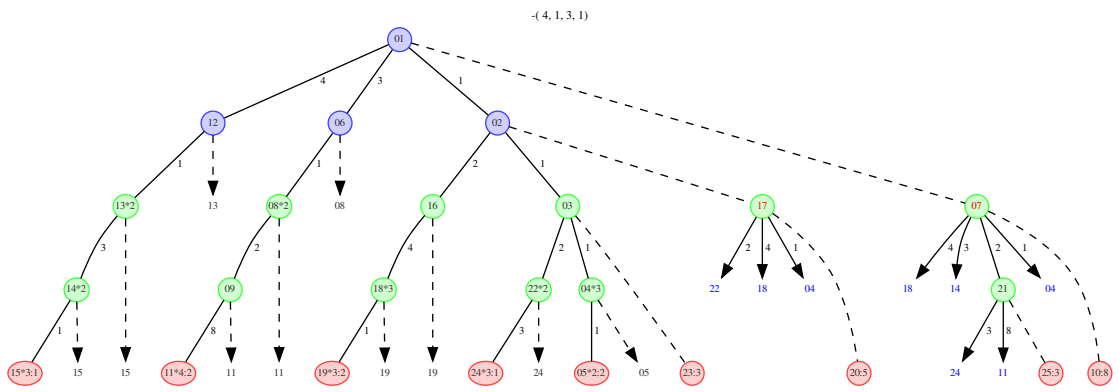
PROPER CHILD, consegue criar arestas SHARED CHILD para seus vértices destino. Note que todos os CIs foram atualizados.

9.2.4 Sequência de Remoções

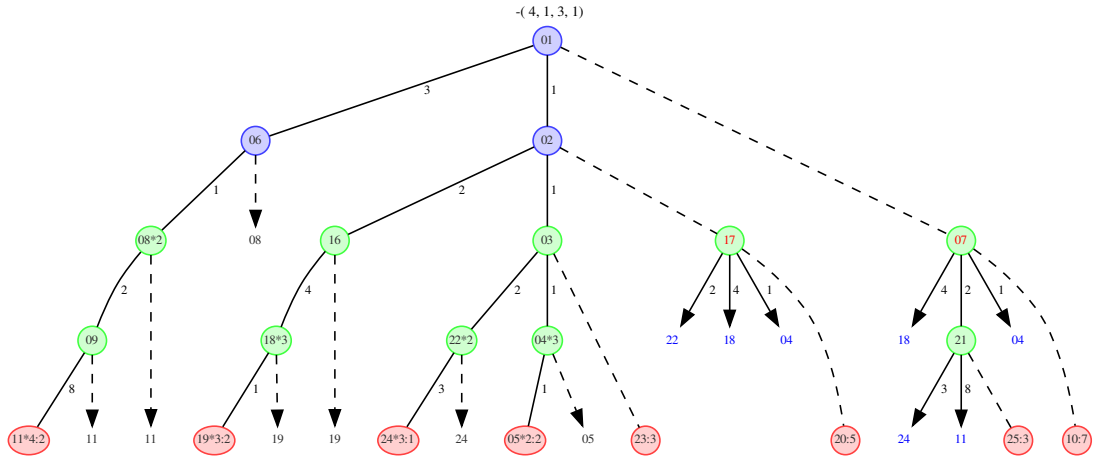
A Figura 9.10 apresenta duas remoções seguidas no endereço (4, 1, 3, 1). Após a primeira remoção (Figura 9.10a), o CI do vértice 15 volta a ser um. Após a segunda (Figura 9.10b), ao CI atingir o valor 0, o processo de remoção que afeta a estrutura é iniciado. Todo o caminho iniciado na aresta com valor 4 é removido, o CI no vértice 10 é atualizado e a aresta SHARED CHILD que apontava para o vértice 14 é removida.

A Figura 9.11 apresenta o resultado da remoção nos endereços (1,1,1,1) e (3,1,2,1), (1,2,4,1) fazendo os CIs dos terminais na tinytree principal assumirem o valor 1. Quando o CI tem o valor 1, a remoção seguinte inicia o processo de remoção estrutural.

A Figura 9.12 apresenta o resultado de quatro remoções. Em (a), vê-se que a remoção no endereço (1, 1, 1, 1) elimina o vértice 04 e todas as arestas SHARED CHILD que apontavam para ele. Em (b), nota-se que a remoção no endereço (3, 1, 2, 8) acaba com a bifurcação e provoca a eliminação da tinytree mais à direita. Em (c), tem-se que a remoção (1, 1, 2, 3) elimina a última tinytree. Finalmente, em (d) a remoção (1, 2, 4, 1) faz o tinycube voltar a conter apenas o vértice Maxiroot.



(a) CI no vértice terminal 15 tem valor 1



(b) Eliminação do vértice terminal “15” porque CI atingiu valor zero

Figura 9.10: Consequências de duas remoções que zeraram o CI de um vértice terminal

9.3 Resultados com datasets sintéticos

Os experimentos de validação demonstraram que os algoritmos de inserção e de remoção funcionavam adequadamente em condições planejadas. No entanto, algumas sequências de operações de inserção e remoção com diferentes valores de endereços poderiam fazer os algoritmos apresentarem peculiaridades ou inconsistências. Por conta deste risco, foram realizados experimentos de teste que submeteram os algoritmos do Tinycubes a diversas sequências de inserções e remoções, utilizando diferentes configurações de dimensões, alturas dimensionais e intervalos de valores.

Para que esses teste de estresse fossem realizáveis, foram criadas diferentes sequências

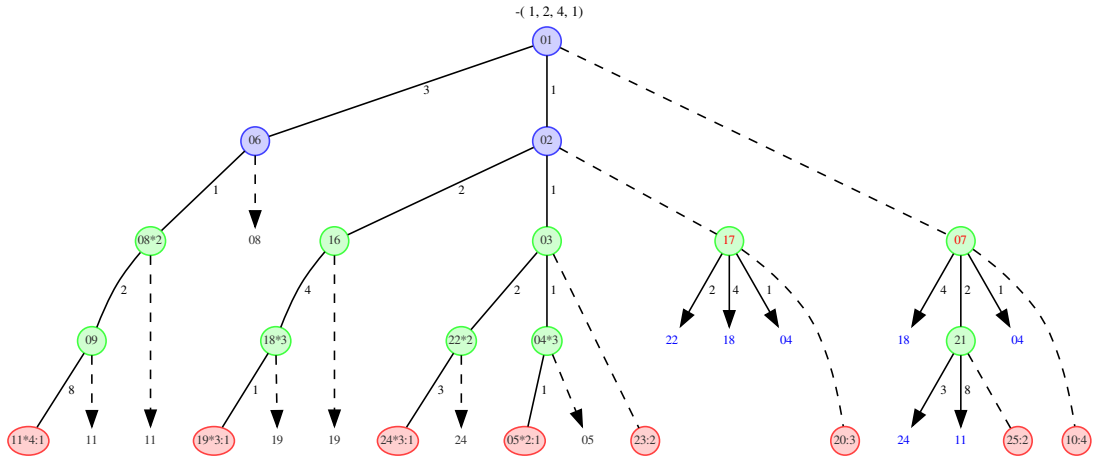
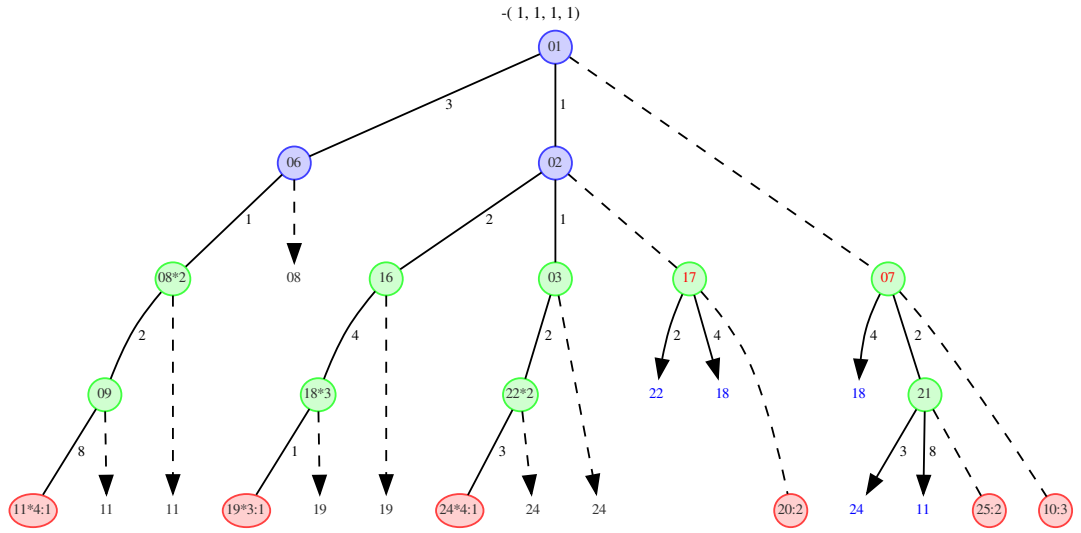


Figura 9.11: Remoçao tripla afetando apenas CIs

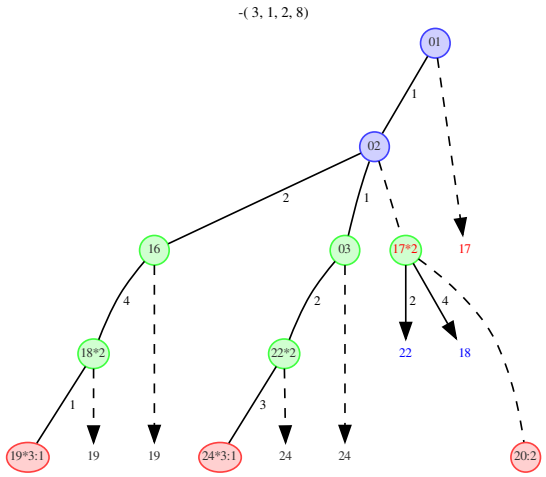
de código na linguagem C, que posteriormente foram embutidas na implementação da estrutura. As sequências de código a s destacar são: uma rotina capaz de gerar arquivos especiais contendo dados sintéticos aleatórios e solicitações de operações de inserção e remoção foi criada; uma rotina que executa as operações de acordo com esses arquivos especiais de dados sintéticos também foi criada; os algoritmos de inserção e remoção foram aparelhados com código para monitoramento, gerando valores coletáveis que poderiam ser analisados em caso de falha; rotinas que realizam a verificação da consistência da estrutura durante a execução.

O procedimento de verificação da consistência da estrutura consiste em avaliar, para todos os vértices v, se a a soma de todos os descendentes de v é igual ao valor obtido à partir da aresta CONTENT de v. Como a execução deste procedimento é recursivo e pode exigir visitar o mesmo vértice diversas vezes, o procedimento de teste permite configurar uma verificação de checagem a cada n operações de inserção ou remoção, onde n pode varia de zero (não executar) até 5000, para grandes volumes de dados.

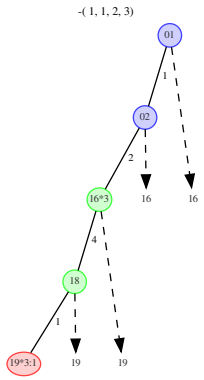
A implementação, contendo todo este código adicional, foi submetida a testes com configurações da estrutura contendo até 12 níveis, subdivididos em diferentes dimensões e alturas dimensionais. Nenhuma anormalidade foi detectada em todos os testes realizados.



(a) Eliminação da aresta SHARED CHILD com valor 1 no vértice 7



(b) Eliminação da tinytree filha do vértice 1



(c) Eliminação da tinytree filha do vértice 2

(d) Maxi-root

Figura 9.12: Remoções eliminando SHAREDs CHILD e tinytrees

9.4 Resultados com datasets reais

Foram analisados dois datasets contendo dados coletados em atividades reais. O primeiro deles é conhecido como "BrightKite" e foi um dos datasets utilizados no artigo que definiu o Nanocubes [38]. Esse dataset contém mais de 4 milhões de registros e foi utilizado neste trabalho como forma de permitir uma comparação do desempenho do Tinycubes

Níveis	D	$\mathcal{H}$	Distinct	Inserções	Remoções	Repetições
2	1	[2]	2	100	0	10
2	1	[2]	2	100	100	10
2	2	[1,1]		1.000	0	10
2	2	[1,1]	2	1.000	1.000	10
8	2	[4,4]	3	100.000	0	10
8	4	[2,2,2,2]	3	1.000.000	0	10
8	4	[2,2,2,2]	3	1.000.000	1.000.000	1
10	2	[8,2]	2	1.000.000	0	1
10	2	[8,2]	2	1.000.000	1.000.000	1
10	2	[8,2]	2	5.000.000	5.000.000	1

Tabela 9.1: Alguns dos cenários de teste de estresse

com o registrado no artigo do Nanocubes. O segundo dataset possui 1 milhão de registros e contém dados coletados decorrentes do uso de operadoras de telefonia por motoristas de taxi durante sua jornada de trabalho na cidade do Rio de Janeiro. A seguir serão apresentados os resultados obtidos para cada um desses datasets.

As figuras desta seção apresentam comparações entre os valores absolutos registrados por diferentes tecnologias em em diferentes frações do dataset. Nas legendas das figuras foram utilizadas as seguintes abreviações para as tecnologias: "Tiny", "Nano+" e "Nano", respectivamente para "Tinycubes", "Nanocubes+" e "Nanocubes". Em vários casos, os gráficos utilizam a métrica "Melhoria", um valor percentual que indica como os valores absolutos medidos utilizando uma tecnologia "A" foram superiores aos valores absolutos medidos utilizando outra tecnologia "B".

Como regra geral, a fórmula utilizada para o cálculo da Melhoria entre "A x B" apresentada em diversas figuras é dada pela expressão:

$$Melhoria = \frac{(V_B - V_A)}{V_B} \times 100$$

quando valores absolutos menores representam algum tipo de ganho. No entanto, no caso particular de arestas SHARED CHILD, valores absolutos maiores é que representam um ganho. Neste caso, a fórmula utilizada para o cálculo da Melhoria foi[4]:

$$Melhoria = \frac{-(V_B - V_A)}{V_B} \times 100$$

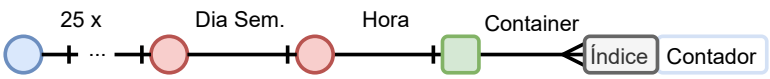


Figura 9.13: Schema Brightkite

9.4.1 Brightkite

O dataset Brightkite está disponível para uso público, e foi formado armazenando registros de "login" na rede social Brightkite, já extinta. Os registros, coletados por Cho et al. [10], contém informações de localização e hora do login, além de outros campos de menor interesse. Os dados foram coletados entre abril/2008 e outubro/2010. O esquema utilizado no experimento com o dataset Brightkite, e representado na Figura 9.13[5], é o mesmo do artigo do Nanocubes: O índice é formado por uma dimensão com 25 níveis derivados de uma *Quadtree* referentes as coordenadas, em latitudes e longitude, do ponto onde o evento ocorreu e uma dimensão complexa de tempo, composta pelo dia da semana e a hora do evento seguidos de um Container, com índice de 16 bits para identificar a hora da medida, que armazena apenas um Content do tipo contador.

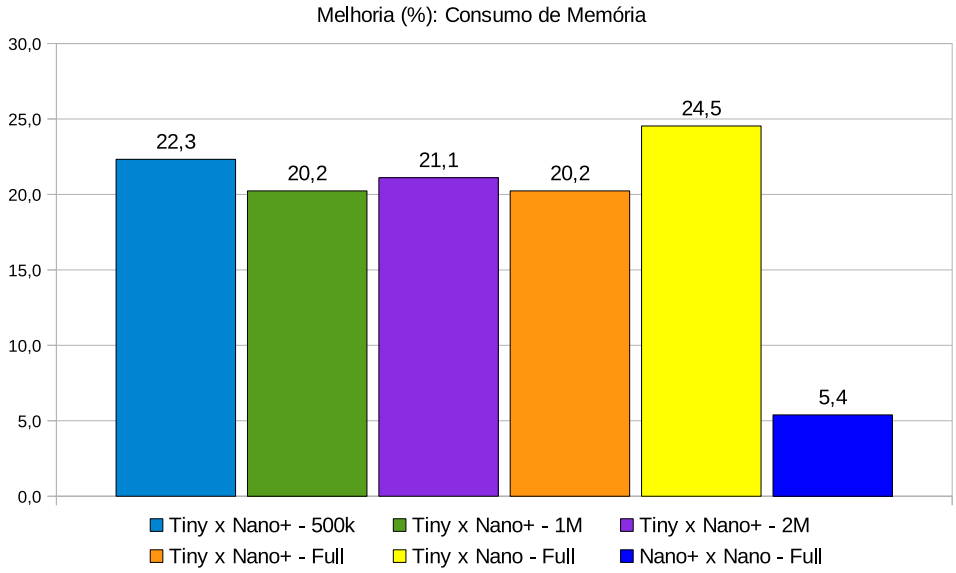


Figura 9.14: Brightkite - melhoria no consumo de memória

Como particularidade deste esquema, o Módulo Container, "binlist-addonly", utilizado nos experimentos, é uma versão da implementação da mesma estrutura de dados utilizada pelo Nanocubes, uma variação da estrutura "Summed Table Sparse". Como no Nanocubes, esse módulo armazena todos os Contents num array, portanto não sofre o "overhead"

[4]O módulo da diferença não foi utilizado no numerador das fórmulas porque poderia mascarar a piora em alguma relação entre as tecnologias.

[5]Esta figura utiliza a mesma legenda da Figura 8.1

decorrente do uso de ponteiros de 64 bits. Por outro lado, o uso de um array cria enormes dificuldades para alocação de dados adicionais após o início da operação, tornando a implementação deste módulo "add-only". Note que, devido ao sistema de modularização do Tinycubes, basta especificar o nome de outro Módulo Container no arquivo de Schema, como por exemplo "binlist", que volta a ser possível realizar remoções na estrutura.

A Figura 9.14 apresenta a melhoria que a implementação do Tinycubes oferece quando comparada às implementações Nanocubes+ e Nanocubes. Cada cor na figura identifica uma fração do dataset Brightkite e qual tecnologia foi utilizada para comparação. Em quase todos os casos, a tecnologia comparada com o Tinycubes foi a implementação Nanocubes+, porque era a única implementação capaz de realizar testes com frações do dataset. A apuração da Melhoria envolvendo o Nanocubes foi baseada nos valores publicados no artigo de referência do Nanocubes [38].

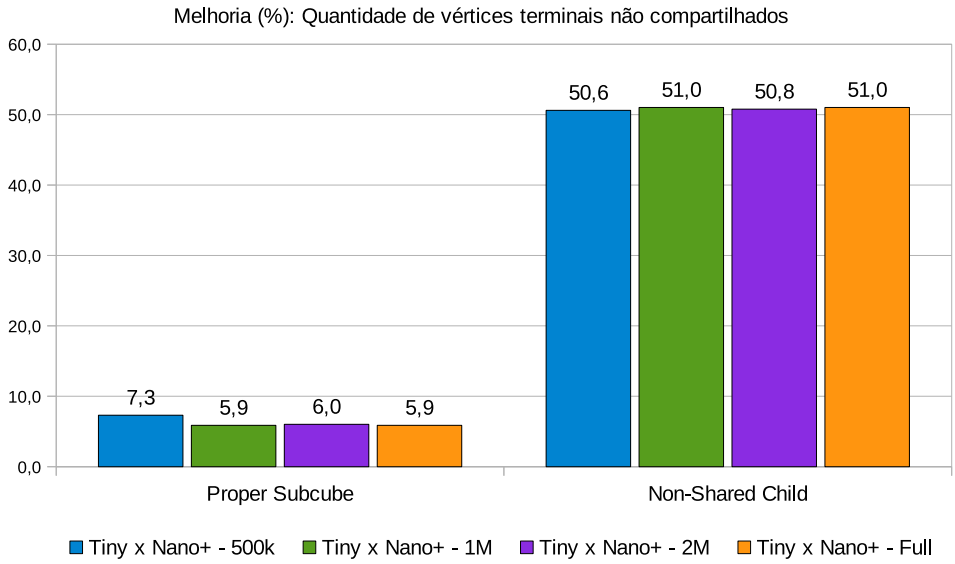


Figura 9.15: Brightkite - Vértices terminais não compartilhados

Como pode-se perceber, a melhoria trazida pelo Tinycubes em relação ao Nanocubes+ mantem-se consistentemente acima de 20% para todas as 4 frações do dataset, embora seja percebido um ligeiro decréscimo à medida que a quantidade de dados aumenta. Quando comparado com os valores publicados no artigo do Nanocubes, a execução do Tinycubes com utilizando o dataset completo apresenta um resultado ainda melhor (em amarelo) do quando comparado com o Nanocubes+. Esse acréscimo de melhoria pode ser entendido ao se perceber que o Nanocubes+ já apresenta uma melhoria em torno de 5% em relação ao Nanocubes ("Nanocubes+ x Nanocubes - Full").

A Figura 9.15 ajuda a explicar como o Tinycubes consegue apresentar resultados

melhores do que as demais tecnologias ao segregar os vértices terminais em categorias. Nesta figura são apresentadas duas das três categorias utilizadas para a análise do impacto da variação dos vértices terminais no resultado final das tecnologias. A primeira categoria agrupa os vértices terminais apontados por arestas PROPER CUBE. Relembrando que vértices apontados por este tipo de aresta nunca são compartilhados, então a redução do seu número por uma tecnologia é um fator importante para redução do consumo de memória. É justamente o que histograma sobre a legenda "Proper Subcube" exibe. Houve uma redução da ordem de 6% no número de vértices terminais apontados por arestas PROPER CUBE.

Para entender a Figura 9.15, é importante relembrar de três fatores: (i) para diminuir a quantidade de memória utilizada sem afetar significativamente a performance nas consultas, tanto o Nanocubes quanto o Tinycubes esforçam-se para compartilhar internamente partes da sua estrutura através de arestas SHARED; (ii) vértices terminais podem armazenar séries temporais e portanto podem demandar grande quantidade de memória; (iii) quanto utilizado para armazenar séries temporais, a quantidade de memória alocada por um vértice terminal é variável, uma vez que vértices terminais descendentes de vértices mais altos na hierarquia agregam os valores de mais vértices do que os descendentes de vértices mais baixos. Assim, o compartilhamento de vértices terminais pode ser importante para a redução da utilização da memória. Todos esses fatores permitem concluir que quanto menor for a quantidade de vértices terminais utilizados, menor será a quantidade de memória utilizada, porém este impacto nem sempre é proporcional.

A outra categoria de vértices terminais exibida na figura é formada por vértices que são destino de arestas CHILD, porém não são compartilhados (legenda "Non-Shared Child", na figura), o que implica que a memória alocada por eles também não é compartilhada. Considere uma série temporal com milhares de entradas. Quando uma série dessas não é compartilhada com outro vértice, pode ser que esse outro vértice tenha de duplicar todos os Contents armazenados. Lembrando que o algoritmo do Nanocubes (e consequentemente do Nanocubes+) possui peculiaridades que levam a criação de vértices terminais desnecessariamente duplicados, dependendo do dataset utilizado, existe uma grande possibilidade da ocorrência de duplicações desnecessárias de séries temporais. Observe que houve uma redução consistente de 50% na quantidade de vértices terminais "Non-Shared Child" em todas as frações do dataset.

Assumindo não houve prejuízo no acesso a valores pré-computados, as duas reduções na quantidade de vértices terminais sinalizam uma maior eficiência da tecnologia

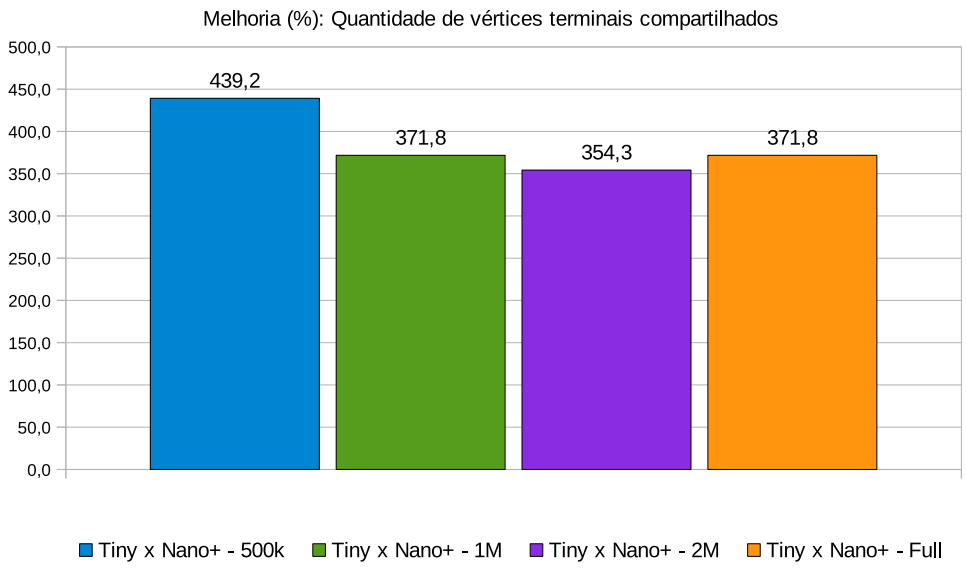


Figura 9.16: Brightkite - Vértices terminais compartilhados

em compartilhar partes da estrutura. Porém, ainda é necessário avaliar a terceira categoria de vértices terminais, os vértices terminais apontados por arestas CHILD que são compartilhados.

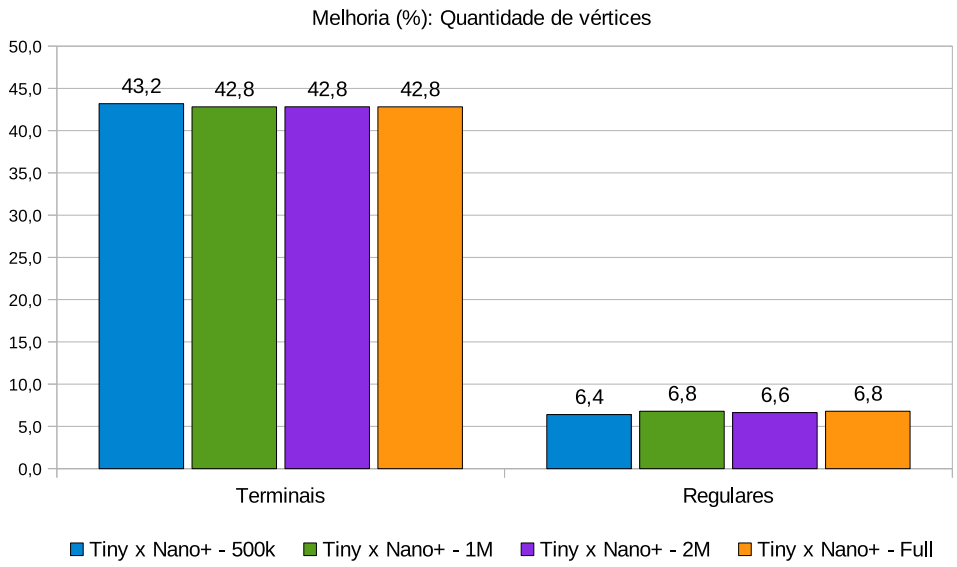
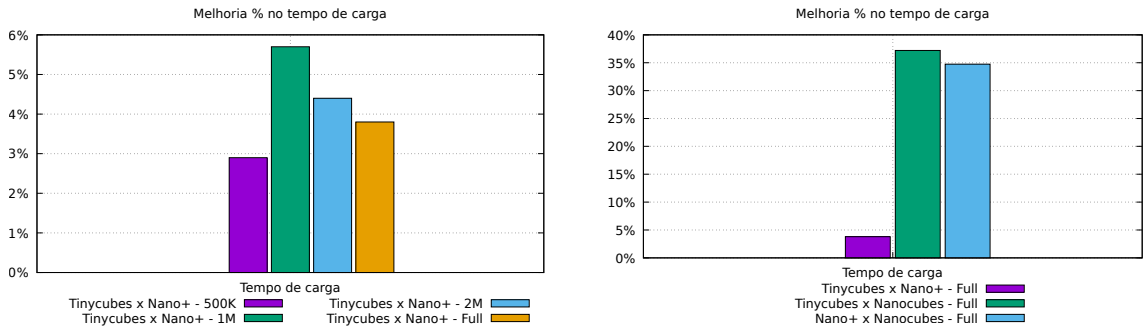


Figura 9.17: Brightkite - total de vértices

A Figura 9.16 exibe o aumento percentual na quantidade de vértices terminais compartilhados. Diferentemente das categorias anteriores, nas quais a redução do número de vértices era considerado uma melhoria, neste caso, ocorre o oposto. A figura mostra que houve um melhoria percentual muito significativa (entre de 350% e 450%) no número de vértices terminais apontados por arestas CHILD que passaram a ser compartilhados.

Essa melhoria é justamente a consequência esperada da utilização do novo algoritmo de inserção do Tinycubes, uma vez que ele resolve as peculiaridades do algoritmo utilizado pelo Nanocubes. Ao se comparar Figura 9.17 com a Figura 9.14, pode se perceber que o decréscimo na melhoria, correspondente à redução de memória na primeira figura, coincide com o decréscimo na melhoria da quantidade de terminais compartilhados.

A Figura 9.17 exibe a redução percentual no número de vértices terminais e de vértices regulares para cada fração do dataset. Nota-se que, para cada tipo de vértice, a melhoria apresentada foi praticamente a mesma em todas as frações do dataset, o que sugere que os dados coletados possuíam uma distribuição quase uniforme dos valores mensurados. Nota-se também que a redução percentual do número de vértices terminais (que armazenam valores) foi bem mais intensa do que a dos vértices regulares (que não armazenam nada), o que reforça a percepção de que a grande redução obtida na utilização da memória do Tinycubes frente as demais tecnologias, deve-se ao alto grau de compartilhamento de vértices terminais.



(a) (b)

Figura 9.18: Brightkite - Tempo de carga

A Figura 9.18(a) exibe o resultado dos experimentos para apuração da melhoria do tempo de carga de cada fração do dataset, comparando o Tinycubes com o Nanocubes+. Esses experimentos utilizaram o mesmo equipamento e as mesmas configurações de hardware. As pequenas melhorias registradas sugerem que os poucos passos adicionais do algoritmo do Tinycubes são mais do que compensados pela redução da quantidade de vértices a serem manipulados.

A Figura 9.18(b) exibe a melhoria calculada utilizando todo o dataset, no entanto, diferentemente dos demais experimentos referentes à carga de dados, a comparação com o Nanocubes foi baseada no tempo informado no artigo de referência [38]. Contudo, nesse artigo, não foi encontrada nenhuma informação referente ao equipamento utilizado para apuração do tempo de carga. Portanto, não é possível garantir que a melhoria

apresentada pela implementação do Tinycubes é realmente 35% mais rápida usando o mesmo equipamento. Ainda considerando a parte (b), vê-se que a implementação do Nanocubes+, que utiliza o mesmo algoritmo do Nanocubes, apresenta uma melhoria da ordem de 35%, o que reforça a hipótese de que essas melhorias decorrem da diferença do hardware utilizado[6].

	BrightKite (0,5M)			BrightKite (1M)			BrightKite (2M)			BrightKite (Full)			BrightKite (Full)		
Métrica	Tiny	Nano+	(%)	Tiny	Nano+	(%)	Tiny	Nano+	(%)	Tiny	Nano+	(%)	Tiny	Nano	(%)
Número de Itens (K)	500	500		1.000	1.000		2.000	2.000		4.747	4.747		4.747	4.747	
Número de Vértices	1.929.725	2.650.173	**27,2**	3.693.902	5.017.170	**26,4**	7272891	9.888.333	**26,4**	15.425.776	20.946.603	**26,4**			
Nº de Vértices Terminais	850.403	1.497.085	**43,2**	1.601.018	2.781.126	**42,4**	3096204	5.415.243	**42,8**	6.504.180	11.374.700	**42,8**			
Nº de Vértices Regulares	1.079.322	1.153.088	**6,4**	2.092.884	2.236.044	**6,4**	4176687	4.473.090	**6,6**	8.921.596	9.571.903	**6,8**			
Terminais: Proper Subcube	237.628	256.377	**7,3**	457.130	487.372	**6,2**	904.202	962.104	**6,0**	1.944.230	2.065.595	**5,9**			
Terminais: Non-Shared Child	612.775	1.240.708	**50,6**	1.143.888	2.293.754	**50,1**	2192002	4.453.139	**50,8**	4.559.950	9.309.105	**51,0**			
Terminais: Shared Child	547.626	101.557	**439,2**	1.048.570	233.832	**348,4**	2055916	452.549	**354,3**	4.330.950	918.003	**371,8**			
Tempo de carga (ms)	10.739	11.061	2,9	22.891	24.279	5,7	53.570	56.023	4,4	131.889	137.056	3,8	131.889	210.000	**37,2**
Memória (MiB)	138,8	178,7	**22,3**	271,8	345,2	**21,3**	543,8	689,3	**21,1**	1.207,4	1.513,7	**20,2**	1.207,4	1.600,0	**24,5**

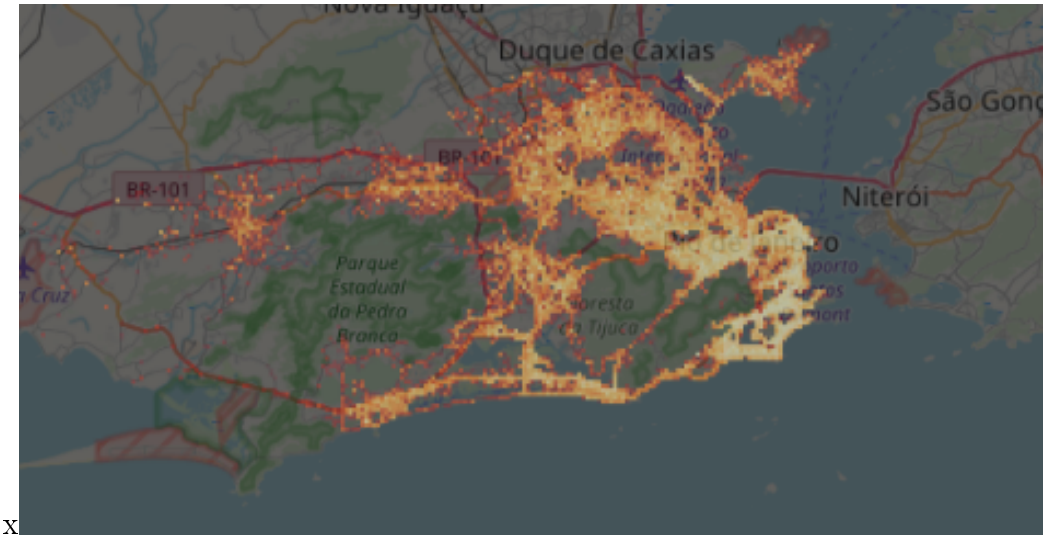
Figura 9.19: Brightkite - valores absolutos

A Figura 9.19 apresenta uma tabela com os valores absolutos apurados nos experimentos e que foram utilizados para compor os gráficos exibidos nesta seção. Nesta tabela, os valores em negrito nas colunas "(%)" correspondem aos valores exibidos como Melhorias. Na tabela existem cinco grupos de três colunas, cada um correspondente a uma fração da carga do dataset Brigthkite, à exceção do último, no qual alguns valores foram obtidos diretamente do artigo do Nanocubes. Os nomes das colunas que identificam a tecnologia são abreviaturas dos nomes reais, onde "Tiny", "Nano+" e "Nano" referem-se, respectivamente, às tecnologias "Tinycubes", "Nanocubes+" e "Nanocubes". A maioria das métricas são auto-explicativas e correspondem diretamente aos valores já exibidos nos gráficos. A única métrica exibida em um gráfico que não consta nesta tabela é a Melhoria do tempo de carga comparando a tecnologia "Nanocubes+" e a "Nanocubes". Esta métrica foi calculada aplicando a fórmula da Melhoria descrita anteriormente com os valores encontrados na linha "Tempo de carga" nas colunas "Nano+" e "Nano" nos grupos com título "Brightkite (Full)".

9.4.2 Operadoras de telefonia utilizadas por táxis

O dataset CelularTaxisRio, ilustrado na figura 9.20, é formado por registros contendo dados referentes ao uso de operadoras de celular por taxistas durante a jornada de trabalho no Rio de Janeiro. Esse dataset contém 1 milhão de registros coletados no período entre 22/06/2016 e 23/06/2016, inclusive. O esquema utilizado no experimento com o dataset,

[6]Deve-se registrar, contudo, que o processador utilizado para os testes do Tinycubes e Nanocubes+ é um processador para desktop lançado no ano de 2012, sendo, portanto, um equipamento contemporâneo aos equipamentos da época em que o artigo foi escrito.



x

Figura 9.20: Heatmap baseado no dataset CelularTaxisRio

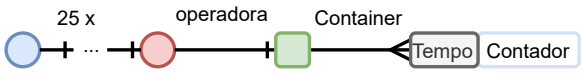


Figura 9.21: Schema CelularTaxisRio

e representado na Figura 9.21[7], é o seguinte: índice formado por uma dimensão com 25 níveis derivados de uma *Quadtree* referentes a latitude e longitude do ponto onde a coleta dos dados ocorreu e uma dimensão simples com um números entre 1 e 5 identificando a operadora de telefonia utilizada. Os vértices terminais são compostos por um Container, com índice de 32 bits para identificar a hora da medida, que armazena apenas um Content do tipo contador.

A Figura 9.22 apresenta a melhoria na utilização da memória que o Tinycubes oferece quando comparado ao Nanocubes+. Cada cor na figura identifica uma fração do dataset CelularTaxisRio. Como pode-se perceber, a melhoria trazida pelo Tinycubes em relação ao Nanocubes+ manteve-se em torno de 4% para todas as 4 frações do dataset, embora seja percebido um ligeiro decréscimo à medida que a quantidade de dados aumenta. Esta faixa de melhoria de 4% é bem inferior à melhoria na casa dos 20% apresentada pelo dataset Brightkite. Os gráficos nas figuras a seguir ajudarão a entender a causa.

Devido à grande queda na melhoria da utilização de memória apresentada por este dataset, é conveniente apresentar previamente os valores absolutos apurados nos experimentos para auxiliar nas explicações. A Figura 9.23 exibe uma tabela com os valores absolutos apurados durante os experimentos. A composição desta tabela é idêntica à da tabela na Figura 9.19, com a diferença que os valores foram medidos à partir do dataset

[7]Esta figura utiliza a mesma legenda da Figura 8.1

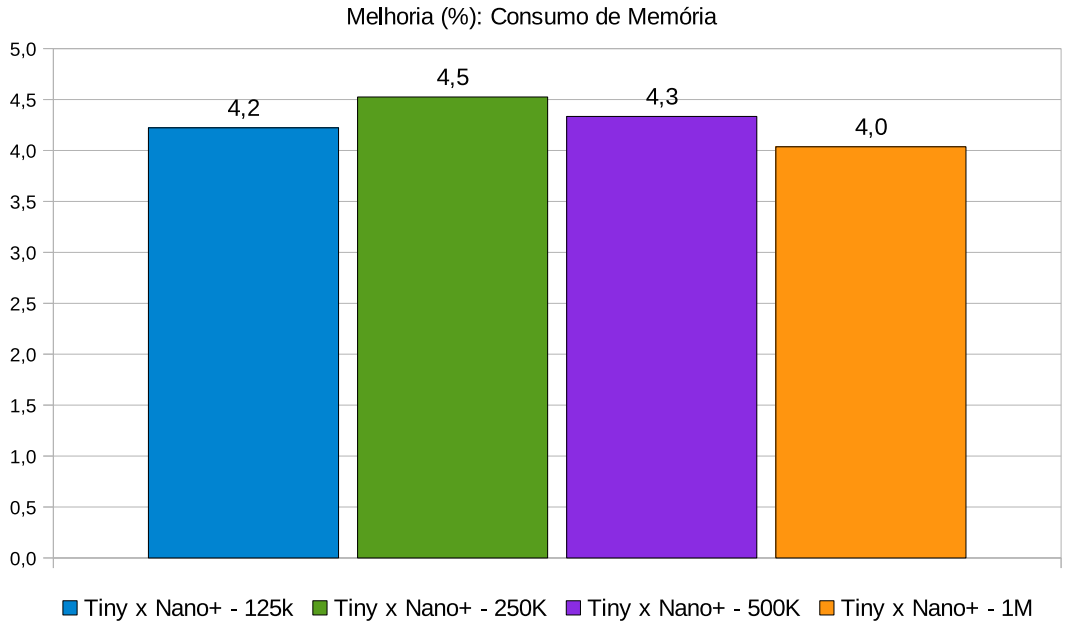


Figura 9.22: CelularTaxisRio - melhoria no consumo de memória

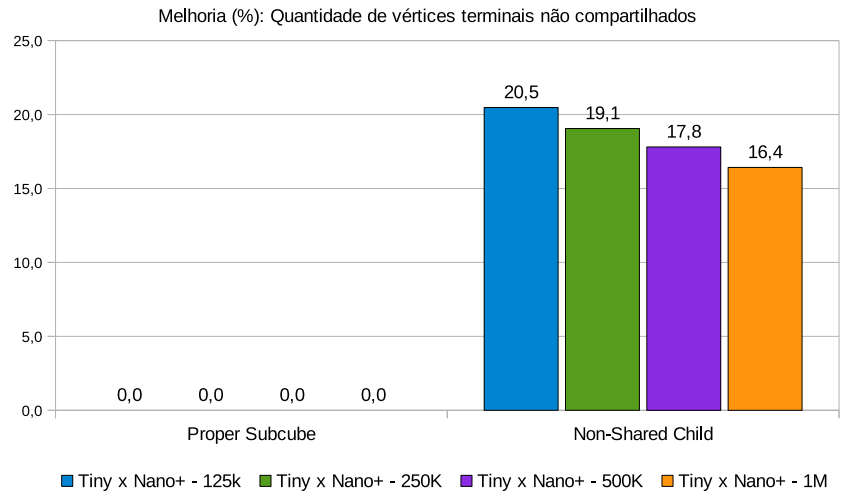
	CelularTaxisRio (125K)			CelularTaxisRio (250K)			CelularTaxisRio (500K)			CelularTaxisRio (1M)		
Métrica	Tiny	Nano+	(%)	Tiny	Nano+	(%)	Tiny	Nano+	(%)	Tiny	Nano+	(%)
Nº de Itens (K)	125	125		250	250		500	500		1.000	1.000	
Nº de Vértices	166.327	174.563	4,7	301.112	316.489	4,9	492703	518.058	4,9	734.768	771.929	4,8
Nº de Vértices Terminais	42.670	50.906	16,2	87.645	103.022	14,9	156571	181.926	13,9	251.219	288.380	12,9
Nº de Vértices Regulares	123.657	123.657	0,0	213.467	213.467	0,0	336132	336.132	0,0	483.549	483.549	0,0
Terminais: Proper Subcube	10.678	10.678	0,0	22.318	22.318	0,0	39.544	39.544	0,0	62.153	62.153	0,0
Terminais: Non-Shared Child	31.992	40.228	20,5	65.327	80.704	19,1	117027	142.382	17,8	189.066	226.227	16,4
Terminais: Shared Child	12.978	4.742	173,7	23.358	7.981	192,7	37137	11.782	215,2	52.217	15.056	246,8
Tempo de carga (ms)	854	867	1,5	1.865	1.902	1,9	6.399	6.449	0,8	7.235	7.416	2,4
Memória (MiB)	9,2	9,6	4,2	17,3	18,1	4,5	30,9	32,3	4,3	52,3	54,5	4,0

Figura 9.23: CelularTaxisRio - valores absolutos

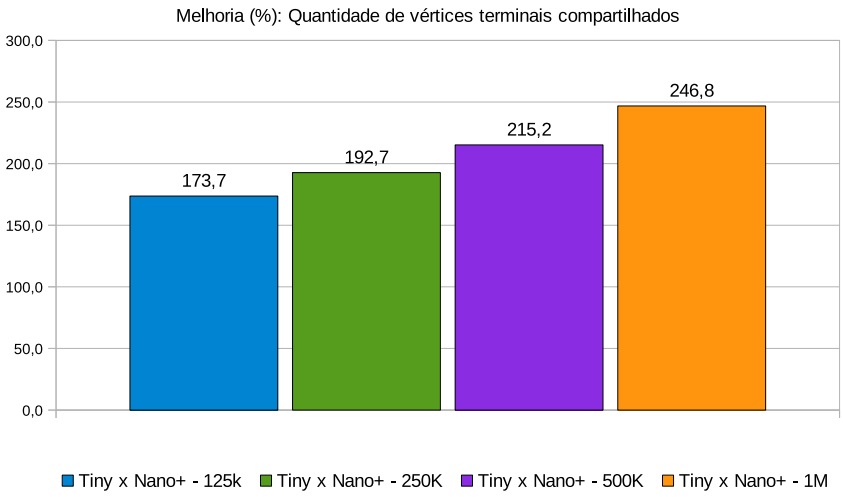
CelularTaxisRio e que não existem dados referentes à tecnologia Nanocubes.

A Figura 9.24(a) apresenta a melhoria na quantidade de Vértices terminais não compartilhados. Como discutido na seção 9.4.1, a redução no número de vértices terminais não compartilhados é considerada uma melhoria por ajudar a reduzir a quantidade de memória consumida pelo sistema. Na figura, observa-se que não houve nenhuma redução no número de vértices terminais apontados por arestas PROPER CUBE. Comparado com o resultado do dataset Brightkite, no qual houve uma redução de 8% no número de vértices dessa categoria, não haver redução, contribui para a baixa melhoria da utilização da memória.

Para entender a razão para esta falta de melhoria nessa categoria de vértices terminais, é necessário relembrar alguns fatos: (i) um vértice terminal apontado por uma



(a) Vértices terminais não compartilhados



(b) Vértices terminais compartilhados

Figura 9.24: CelularTaxisRio - Vértices Terminais

aresta PROPER CUBE é filho de um vértice na dimensão anterior a dele; (ii) o Schema utilizado neste dataset define que os vértices terminais estão na dimensão 3; (iii) nesse Schema, é obrigatório criar um vértice terminal y na dimensão 3 se existirem duas arestas CHILD que são filhas do vértice pai x (na dimensão 2) desse vértice terminal y; (iv) o Schema define a dimensão 2 como dimensão simples, logo não existem vértices MID nela; (v) as peculiaridades do algoritmo original tinham o efeito colateral de criar novas arestas PROPER CHILD e consequentemente novos vértices, que só podem ser MID se a dimensão for complexa, o que não é o caso; (vi) as melhorias do algoritmo Tinycubes preservam as informações originais, portanto, não é possível eliminar arestas CHILD; (vii) neste schema, a única maneira do algoritmo do Tinycubes reduzir a criação de vértices

terminais apontados por arestas PROPER CUBE exige a eliminação de vértices pai MID que apontam para essas arestas. Com bases nesses fatos, é possível chegar a seguinte conclusão: como a dimensão anterior à dimensão do vértice terminal é simples, o algoritmo do Tinycubes não tem vértices MID para eliminar, portanto não há possibilidade de redução desta categoria de vértices terminais.

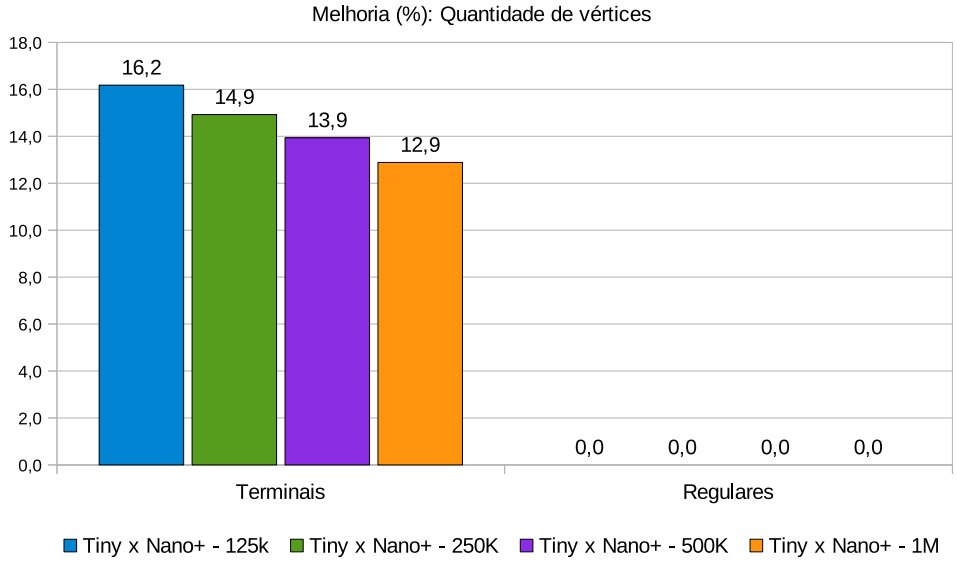


Figura 9.25: CelularTaxisRio - total de vértices

A Figura 9.24(a) também exibe uma redução percentual no número de vértices terminais apontados por arestas CHILD que não são compartilhados. Novamente, o valor desta redução é aproximadamente 50% inferior à redução apresentada no dataset Brightkite. Uma das hipóteses possíveis é o fato da dimensão 2 ser simples, o que oferece menos oportunidades para otimização de arestas CHILD antes de se chegar a um vértice terminal. No entanto, o exame da Figura 9.25, mostra que o perfil de redução de vértices terminais não compartilhados é idêntico ao perfil de redução de todas as categorias de vértices terminais em geral. De fato, ao se examinar a tabela na Figura 9.23 pode-se constatar que toda redução no número de vértices terminais veio da redução de vértices terminais apontados pr arestas CHILD que não eram compartilhados.

A Figura 9.24(b) exibe a melhoria (aumento) percentual na quantidade de vértices terminais apontados por aresta CHILD que são compartilhados. Mais uma vez, o valor desta melhoria é aproximadamente 50% inferior à melhoria apresentada no dataset Brightkite, porém ela tem um perfil diferente. Enquanto a melhoria no dataset Brightkite começava intensa e depois oscilava em torno de 370%, a melhoria no dataset CelularTaxisRio apresenta um perfil crescente. Isso sugere que, à medida que o número de amostras

aumenta, o algoritmo Tinycubes encontra mais oportunidades de otimização.

A Figura 9.25 exibe a redução percentual no número de vértices terminais e de vértices regulares. Destaca-se na figura o fato de não ter havido nenhuma redução na quantidade de vértices regulares. Isso decorre do fato do algoritmo Tinycubes apresentar melhorias ao otimizar o uso de arestas SHARED CHILD em subtrees e, neste Schema, as arestas CHILD das subraízes apontam diretamente para vértices terminais, não existindo nenhum vértices regular a compartilhar. A figura também exibe a redução percentual da quantidade total de vértices terminais. Conforme mencionado anteriormente, toda redução nesse tipo de vértice se originou da redução dos vértices não compartilhados apontados por arestas CHILD.

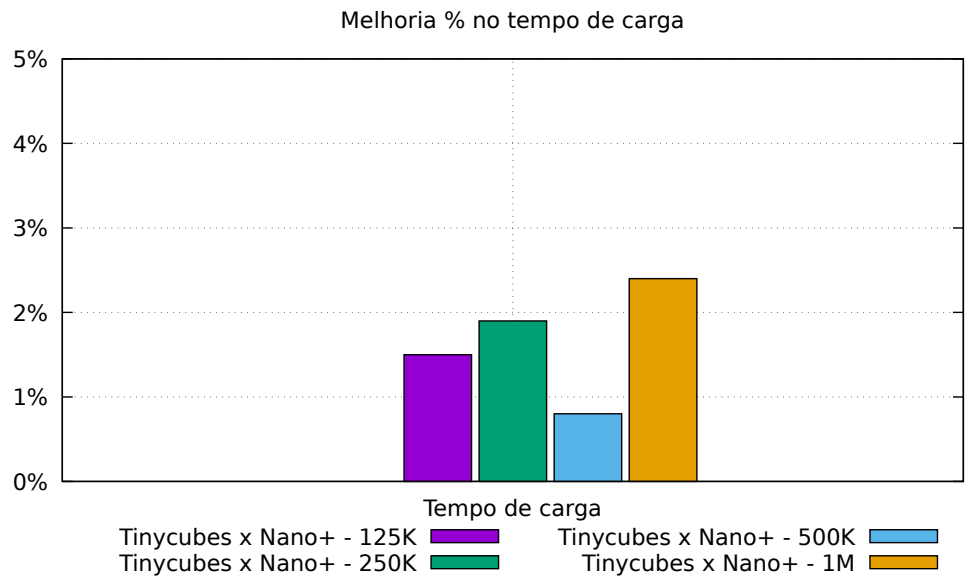


Figura 9.26: CelularTaxisRio - Tempo de carga

A Figura 9.26(a) exibe o resultado dos experimentos para apuração da melhoria percentual do tempo de carga de cada fração do dataset, comparando o Tinycubes com o Nanocubes+. Houve uma ligeira melhoria nesta métrica, sugerindo que o algoritmo do Tinycubes não foi capaz de reduzir significativamente a quantidade de vértices a serem atualizados durante a inserção. Porém, dado o número relativamente pequeno de elementos no dataset, pode-se ver na tabela da Figura 9.23, que o tempo absoluto de carga também é relativamente baixo, na ordem de 1000 ms, o que torna possível que pequenos ganhos de eficiência durante a carga dos dados, seja mascarado pela ação do escalonador de tarefas do computador.

9.4.3 Conclusão

O algoritmo de inserção do Tinycubes demostrou ser mais eficiente do que o algoritmo do Nanocubes em relação ao consumo de memória nos dois datasets reais analisados. No dataset CelularTaxisRio, a redução no consumo de memória foi consistentemente superior à apresentada pelo algoritmo de inserção do Nanocubes[8] para todas as frações do dataset testadas, porém não superou 4,5%. Uma análise so Schema utilizado com esse dataset, revelou que havia poucas oportunidades para superar as peculiaridades do algoritmo do Nanocubes porque as subtrees eram de dimensão simples.

O dataset Brightkite foi utilizado na comparação da implementação do Tinycubes com as implementações Nanocubes+ e Nanocubes, sendo que os resultados utilizados na comparação desta última implementação foram obtidos à partir da valores publicados. A implementação do Tinycubes apresentou uma redução consistente no consumo de memória em todos as comparações realizadas, sempre superior à 20%, apresentando o melhor resultado, quando comparada à implementação do Nanocubes, que chegou a 24,5%[9]. A análise das causas da melhoria obtida pelo algoritmo do Tinycubes revelou que o ganho foi originado na eliminação das peculiaridades do algoritmo do Nanocubes.

Em todas as frações de todos os datasets reais testados, o algoritmo de inserção do Tinycubes sempre apresentou uma eficiência superior à apresentada pelo algoritmo de inserção do Nanocubes, implementado no Nanocubes+, sem que fosse detectado nenhum prejuízo no tempo de execução. Ao contrário, embora pequena, também foi observada uma consistente redução percentual no tempo de execução para todos os cenários testados. Isso permite concluir que a tecnologia Tinycubes confirma as expectativas de ser, diretamente mais eficiente na alocação de memória, e indiretamente mais rápida, no tempo de execução.

[8] implementado no Nanocube+

[9] A melhoria pode ser ligeiramente superior, dado que o Tinycubes utilizou valores com 3 casas decimais de precisão (1.207), enquanto o resultado divulgado pelo Nanocubes foi arredondado para uma casa decimal (1.6), o que pode significar um valor real significativamente maior (1.649) e elevar a melhoria percentual para 26,8%

9.5 Caso de uso - Ferramenta Network Borescope

Esta seção apresenta um caso de uso real da tecnologia Tinycubes para criação da ferramenta Network Borescope que realiza análises de tráfico de redes com auxílio de técnicas de Inteligência Artificial.

9.5.1 Introdução

A ferramenta-protótipo Network Borescope é o resultado de um projeto entre a UFF, a RNP e a Microsoft que teve como objetivo, a aplicação de ferramentas de Inteligência Artifical oferecidas pela Microsoft a problemas de interesse da RNP. Com o intuito de atender aos objetivos do projeto, a tecnologia Tinycubes foi estendida, possibilitando a aplicação de técnicas de IA para obtenção de novas informações a serem disponibilizada durante a exploração visual e interativa. Especificamente para o projeto, as extensões utilizadas foram métodos de IA do framework ML.NET para detecção de peculiaridades e predição de tendências. O projeto foi concluído em outubro de 2019 com a entrega de um protótipo funcional da ferramenta.

A Figura 9.27 ilustra como a ferramenta utiliza a tecnologia Tinycubes para produzir informações sobre contagens, médias, variâncias calculadas à partir de dados reais coletados de roteadores operacionais da RNP.

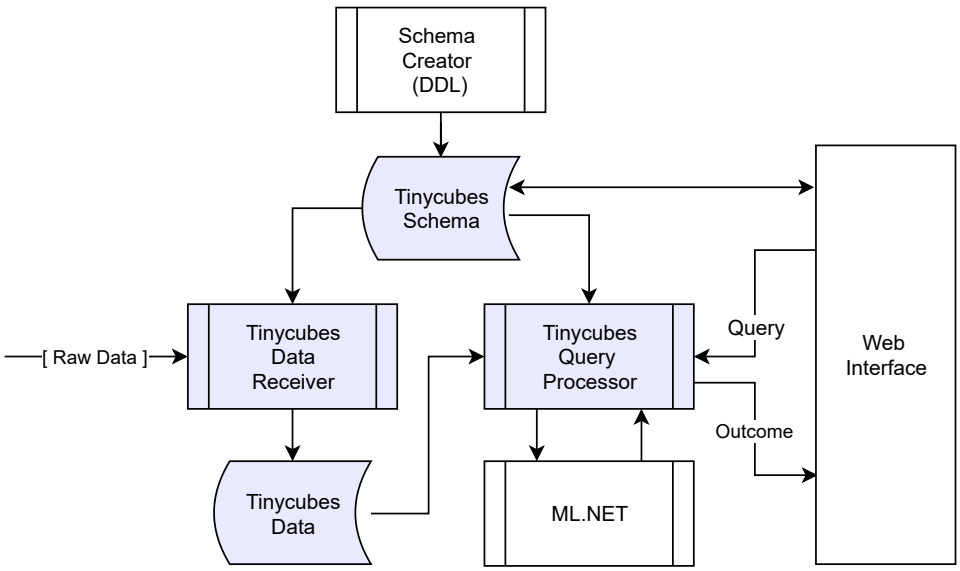


Figura 9.27: Visão esquemática da ferramenta Network Borescope

9.5.2 Dados e Schema

Dados

Os dados utilizados pela ferramenta são resultado da coleta de metadados do tráfego de todos os roteadores da RNP. Esses metadados estão armazenados em arquivos no formato netflow e dispostos como fluxo de dados, onde cada registro contém um ponto de origem, um ponto de destino, os instantes reconhecidos como inicio e final do fluxo, procotolo, eventuais portas do protocolo, bytes e pacotes transferidos, etc.

Como foram identificados diversos fluxos de longa duração, chegando em alguns casos a durar dias, foi necessário transformar cada registro de fluxo em pacotes da dados virtuais, uma vez que a tecnologia Tinycubes foi projetada para lidar com eventos que ocorreram em momentos do tempo, como chegada de pacotes, e não com períodos de tempo variáveis. Desta forma, cada fluxo registrado foi convertido numa sequência de pacotes virtuais distribuídos ao longo do tempo de duração do fluxo. Cada um desses pacotes carregava uma fração do total de dados transmitidos, sendo que a quantidade total desses pacotes virtuais era igual ao total de pacotes do fluxo. Na implementação utilizada, os pacotes virtuais foram distribuídos uniformemente ao longo do tempo e a quantidade de tráfego no fluxo foi distribuída uniformemente por esses pacotes.

A quantidade total de pacotes pacotes virtuais por segundo variou significativamente. No dia 6/8/2019 foram observadas variações entre 2 e 15 mil pacotes virtuais por segundo. Isso significa que a tecnologia deveria ser capaz de processar pacotes nesta taxa para que não fosse necessário descartar pacotes. Durante os experimentos, utilizando o hardware descrito na Seção 9.1, a tecnologia Tinycubes precisou de apenas 6 segundos para processar cada minuto de dados no horário de pico (14:33 hs), apresentando uma capacidade de inserção de aproximadamente 80 mil registros por segundo. Porém deve ser considerado que as coordenadas geográficas utilizadas eram dos pontos de presença da RNP, portanto os dados estavam concentrados em poucos pontos fixos, embora geograficamente afastados entre si.

Schema

O Schema do protótipo da ferramenta permite, além das consultas espaciais e temporais, consultas utilizando o protocolo IP como critério de seleção/agrupamento de informação. Como nos dados não havia informação espacial, no conjunto de dados de entrada foi inserido um campo contendo a sigla do roteador (AM, AP, SP, etc.). A partir da infor-

mação nesse campo, o módulo organizador de dados computava a latitude e longitude e armazenava no Record a ser utilizado.

O Schema utiliza um container do tipo "binlist", indexado pelo tempo do pacote e agrupado em *bins* de 60 segundos de resolução. Esse container continha Contents configurados para produzir uma informação sobre contagem de eventos e também informações sobre diversos detalhes referentes à quantidade de bytes de entrada por pacote (ibytes), como: soma, média, variância, desvio padrão (SD) e maior quantidade de bytes por entrada por pacote.

9.5.3 Interface web ↔ Servidor Tinycubes

Nesta seção, a descrição do funcionamento da tecnologia em um cenário de uso real será baseada em figuras obtidas pela capturas de tela durante o uso da ferramenta.

Descrição da interface web

A Figura 9.28 apresenta a janela inicial da ferramenta onde é possível se identificar três áreas distintas. A parte inferior da imagem é reservada para exibição de um gráfico baseado em dados temporais. Na maior parte da imagem, pode-se ver um mapa da Terra. Sobre esta área de visualização geográfica, existe uma pequena janela para exibição de histogramas, que na figura tem o título "Sum (ibytes) [Log Mb]". Todas essas áreas são capazes de exibir informações baseadas em dados espaciais/geográficos e temporais que estão presentes nas áreas do mapas e do gráfico temporal. As áreas onde são apresentados gráficos convencionais são de fácil entendimento, enquanto a intensidade da cor nos pontos iluminados sobre o mapa representa a intensidade da informação que aquele ponto carrega em relação aos demais.

Listagem 9.1: Limites de exibição

```
1  {"bounds": "hours",
2    "where": [["location", "zrect", 17, 85, -179, -85, 179]],
3    "id": 13}
4  { "bounds": "location",
5    "where": [["location", "zrect", 17, 85, -179, -85, 179]],
6    "id": 14}
7  {"id":13, "tp":200,
8    "result":{"vs": [[1565101800],[1565103719]]},"ms":0 }
9  {"id":14, "tp":200,
10   "result":{"vs": [[8256483,5225849],[9857550,6759664]]},"ms":0 }
```

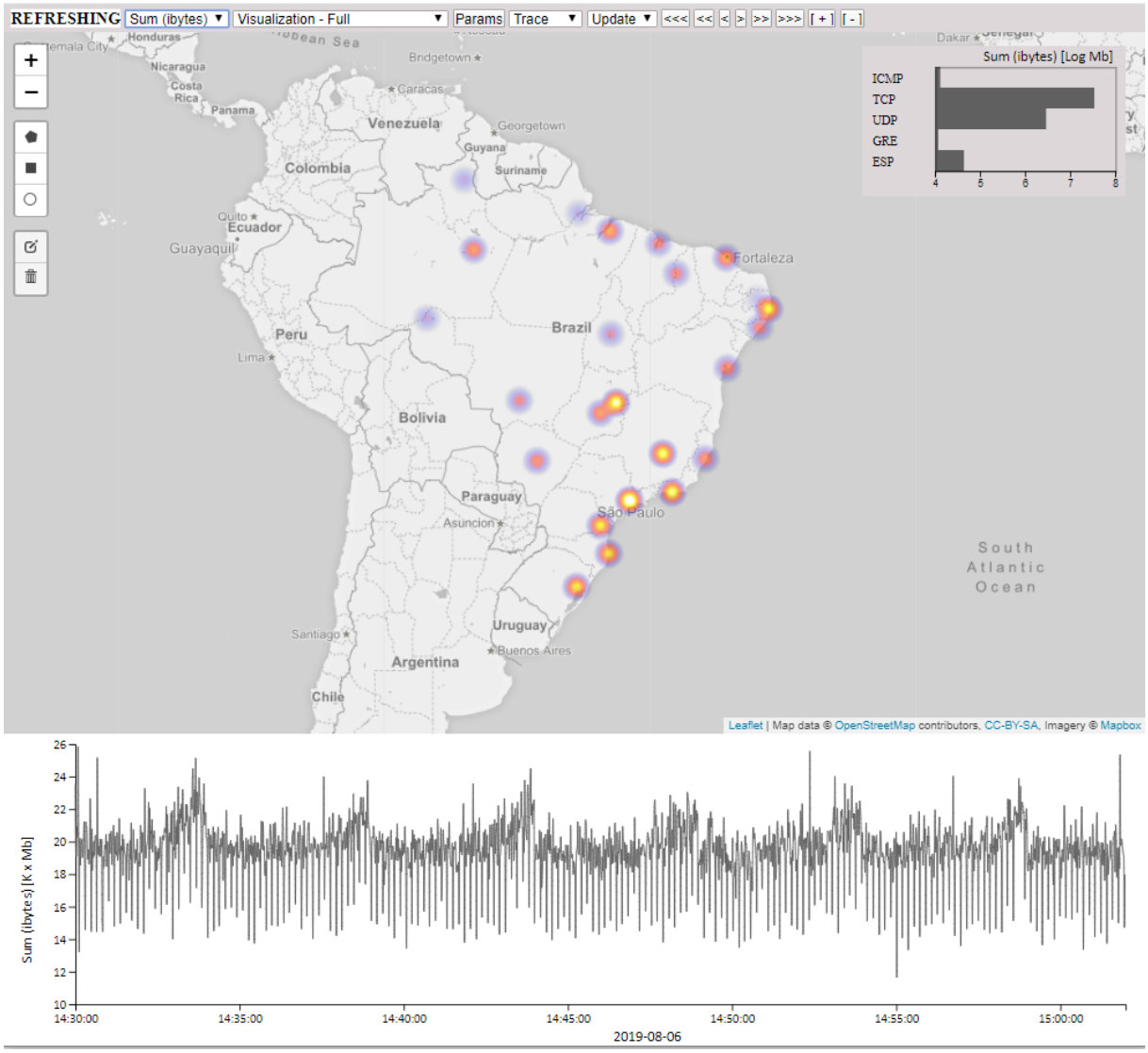


Figura 9.28: Network Borescope

Limites visuais - Bounds

Ao iniciar a execução, a interface necessita descobrir qual parte da Terra deve exibir sobre a área de mapas, bem como identificar um período de tempo a ser exibido no gráfico temporal. Para isso, a interface envia uma consulta "bounds" ao servidor Tinycubes, solicitando limites ("bounds") geográficos e temporais conhecidos que também atendam a cláusula "where" presente na consulta. O servidor, ao receber uma consulta desse tipo, localiza os valores mínimos e máximos do campo especificado em "bounds" que ainda possuam algum Content. Em seguida gera as respostas informando esses limites. Todo este diálogo pode ser visualizado na Listagem 9.1. Observe que o campo "id" é utilizado para identificar a qual consulta a resposta corresponde, uma vez que o protocolo de comunicação AJAX não garante a ordem de envio e recebimento de mensagens.

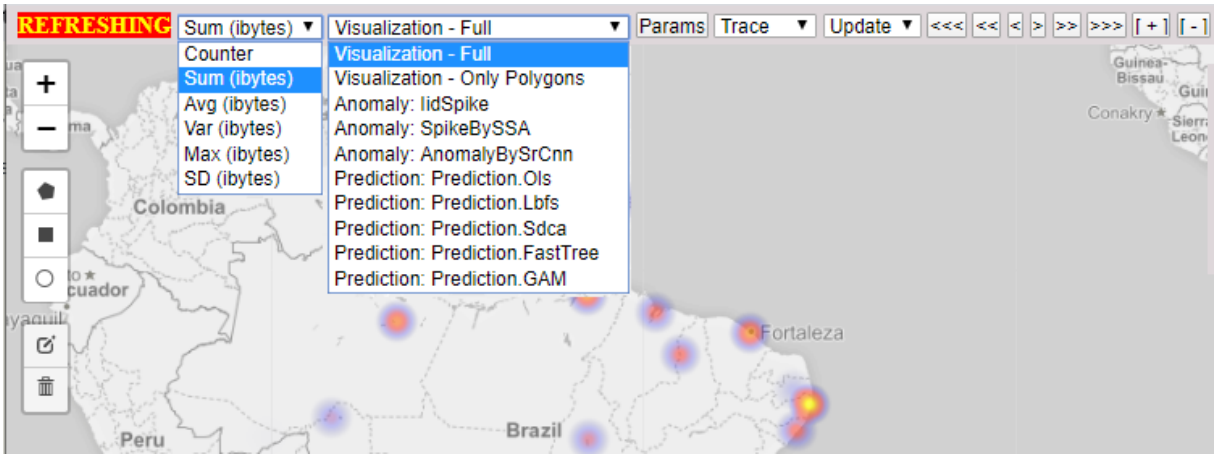


Figura 9.29: Detalhes da interface incluindo os métodos IA

```
{"select": ["hsum"], "group-by": "location", "group-by-output": "kv",
  "where": [
    ["location", "zrect", 8, 14.349547837185375, -72.07031250000001,
      -39.57182223734373, -30.673828125000004],
    [ "hours", "between", 1565101800, 1565101831]],
  "id": 15}
{"select": ["hsum"], "group-by": "hours", "group-by-output": "full",
  "where": [
    ["location", "zrect", 8, 14.349547837185375, -72.07031250000001,
      -39.57182223734373, -30.673828125000004],
    [ "hours", "between", 1565101800, 1565101831]],
  "id": 16}
{"select": ["hsum"], "group-by": "proto", "group-by-output": "kv",
  "where": [
    ["location", "zrect", 8, 14.349547837185375, -72.07031250000001,
      -39.57182223734373, -30.673828125000004],
    [ "hours", "between", 1565101800, 1565101831]],
  "id": 17}
```

Listagem 9.2: Exemplo de consultas enviadas

Consultas - Respostas

A seguir, a interface envia três consultas solicitando informações para serem exibidas sobre o mapa, sobre o gráfico temporal e sobre a área para exibição de informações espaço-temporais. O servidor analisa essas consultas e produz as respostas apropriadas. Na Listagem 9.2, pode-se ver o texto das consultas. Note que o valor do objeto "where" e do campo "select" são os mesmos nas três consultas indicando que a mesma informação básica foi selecionada, no entanto, o campo "group-by" faz com que essa mesma informação básica seja agrupada de forma diferente a cada consulta. Para exibição sobre o mapa ("id": 15), o agrupamento se dá em coordenadas de "location". Para exibição no gráfico temporal

("id": 16), o agrupamento se dá sobre "hours". Para exibição no área espaço-temporal ("id": 17), o agrupamento se dá sobre o valor do protocolo.

A Listagem 9.3 apresenta as respostas para as consultas solicitadas. As respostas estão parcialmente listadas porque são muito longas. Observe que o valor em ms é zero, indicando que cada consulta demorou menos do que 0.5ms para ser realizada.

```
1  {"id":15, "tp":2,
2    "result":[
3      {"k":[125,84],"v":[2693]},{"k":[128,91],"v":[1726]},
4      ...
5      {"k":[150,91],"v":[33314]}],"ms":0 }
6  {"id":16, "tp":4,
7    "result":{
8      "ks":[[1565101800,...,1565101830]],
9      "vs":[[23528,...,20774]]},"ms":0 }
10 {"id":17, "tp":2,
11   "result":[
12     {"k":[1],"v":[206]},{"k":[6],"v":[571600]},
13     ...
14     {"k":[112],"v":[0]}],"ms":0 }
```

Listagem 9.3: Respostas (parte) para as consultas realizadas

Exploração Visual e Interativa de Dados

A Figura 9.30 ilustra como dados coletados enviados para a tecnologia podem ser explorados de forma visual e interativa. A interface web permite que regiões do mapa sejam selecionadas utilizando-se figuras geométricas como retângulos e polígonos. A partir daí, a interface passa a fazer solicitar informações cujos dados originais tinham informações geográficas compreendidas dentro destas áreas. Como resultado, a interface passa a exibir mais informações simultaneamente, permitindo que o operador faça comparações entre diferentes áreas. Nota-se que na janela com histogramas passaram a aparecer informações dependentes das áreas selecionadas.

A Figura 9.31 exibe o resultado da aplicação de um algoritmo de Inteligência Artificial para detecção de anomalias. O gráfico sob o mapa apresenta dois tipos de informação: o resultado de uma consulta solicitando uma lista com a soma de todos os bytes recebidos no período sendo exibido na tela (plotado como um gráfico na cor cinza), e o resultado da detecção de anomalias sobre os elementos desta lista (plotado como pontos amarelos sobre o gráfico).

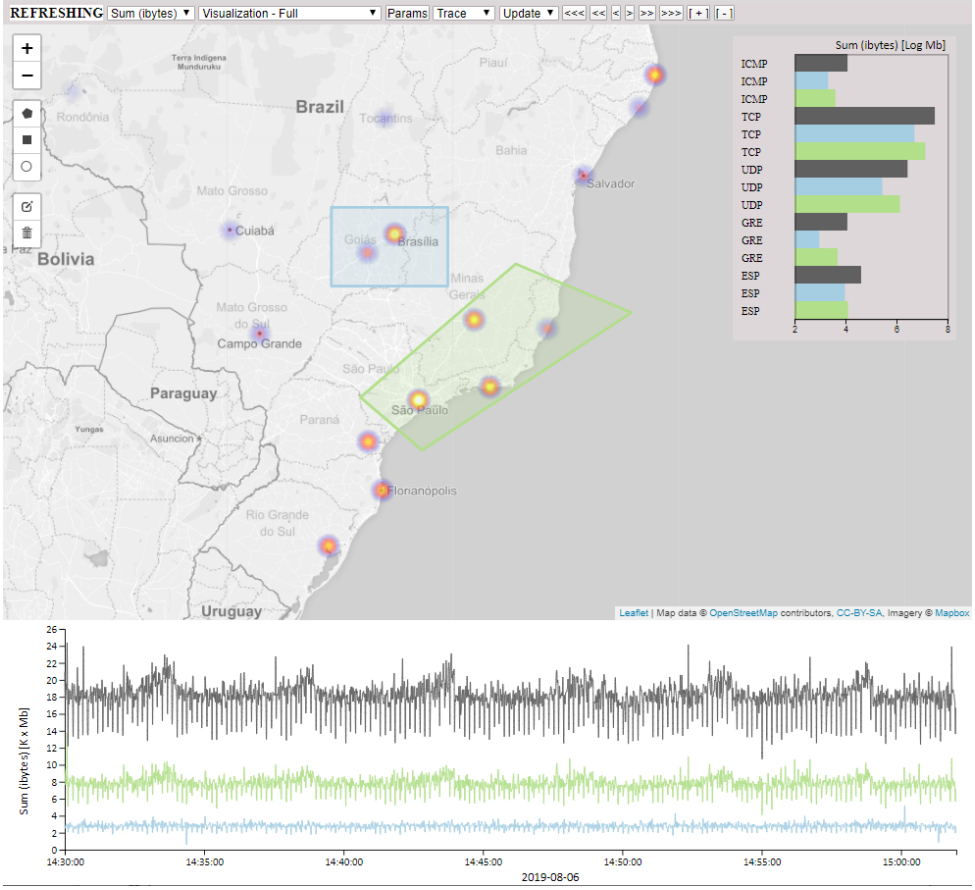


Figura 9.30: Network Borescope - Consultas simultâneas

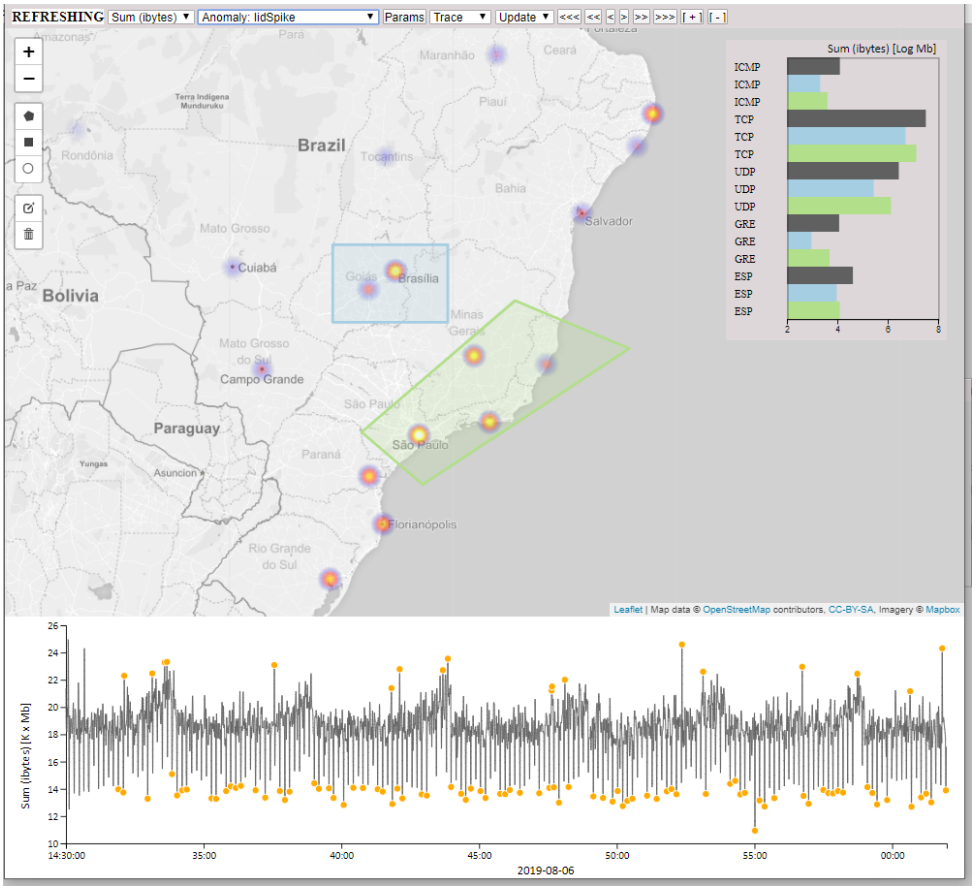


Figura 9.31: Network Borescope - Detecção de Anomalias

Capítulo 10

Conclusão

Este capítulo apresenta as considerações finais sobre o trabalho realizado, descreve contribuições identificadas e relaciona limitações e uma relação de possíveis trabalhos futuros a serem realizados.

10.1 Considerações finais

A tecnologia Tinycubes apresentada neste trabalho oferece uma solução computacional modular que permite a exploração visual interativa de grandes volumes de dados espaço-temporais multidimensionais gerados continuamente, alcançando o objetivo principal proposto para este trabalho. Ao mesmo tempo, a solução desenvolvida atende a todos os requisitos demandados inicialmente (Seção 1.1). Ao se utilizar a estrutura Nanocubes como base, foi possível preservar os requisitos já atendidos por essa estrutura. Através da execução das atividades de pesquisa relacionadas na Seção 1.2, foi possível atender aos demais requisitos demandados.

10.2 Contribuições

Este trabalho apresenta algumas contribuições referentes ao gerenciamento de dados para exploração visual interativa de dados (EVID) baseado em datacubes e, em especial, derivadas da estrutura Nanocubes.

Do ponto de vista teórico, este trabalho apresentou as seguintes contribuições:

- A definição formal da estrutura de dados que suporta a exploração visual e interativa de dados como um Grafo, o que possibilita utilizar conhecimentos oriundos da Teoria

dos Grafos em análises teóricas;

- A criação de um modelo matemático para as operações associadas à extração de informações exatas de um Tinycubes (e de um Nanocubes), definindo formalmente requisitos que devem ser atendidos.

A estrutura de dados Tinycubes apresenta quatro características inovadoras em relação à estrutura Nanocubes:

- Redução do número de vértices necessários para implementação de um datacube;
- Esquema de indexação otimizado em relação ao Nanocubes, identificando mais possibilidades de compartilhamento de partes da estrutura e potencialmente[1] reduzindo a quantidade de memória utilizada;
- Criação de um algoritmo para remoção de dados (desfazer inserções) durante a operação;
- Especificação dos algoritmos utilizados pela estrutura de modo que sejam claramente adaptáveis a novos tipos de dados e informações.

A tecnologia Tinycubes, entendida como os elementos computacionais para além da estrutura de dados, apresenta algumas contribuições:

- Criação de uma linguagem de consulta para EVID similar à linguagem SQL;
- Criação de um processador de consultas para essa linguagem;
- Criação de uma interface genérica de visualização de dados geo-temporais;
- Definição de uma sistema modular para utilização de novas categorias de dados;
- Definição de uma sistema modular para extração de novos tipos de informação;
- Definição de uma sistema modular para utilizar novas formas de organização de Contents(Containers).

[1] a redução real depende da implementação

10.3 Limitações e Trabalhos Futuros

Foram identificadas algumas possibilidades para o desenvolvimento de novos trabalhos à partir dos resultados obtidos nesta dissertação. Esses novos trabalhos variam desde melhorias na implementação propriamente dita até o desenvolvimento de mais estudos formais sobre a estrutura. Nesta seção relacionamos algumas dessas possibilidades.

Otimização interna da estrutura

Os processadores comercializados atualmente têm capacidade para utilizar, no máximo, 128 GiB bytes de memória, ou seja, não utilizam mais do que 37 bits para endereçar toda a memória física (o uso de paginação ou outra forma de memória virtual provocaria um impacto severo na performance). No entanto, a implementação atual aloca ponteiros de 64 bits, o que representa um desperdício de memória de mais de 40% para cada ponteiro alocado. Como boa parte da memória alocada pela estrutura Tinycubes é utilizada por ponteiros, o desperdício total de memória é relevante. Considerando que a disponibilidade de memória é crucial para que mais dados possam ser analisados, pode ser interessante pesquisar formas mais econômicas para armazenamento dos dados.

Materialização seletiva de tinytrees

Cada tinytree originada em aresta PROPER CUBE utiliza recursos computacionais para sua criação e manutenção. Cada aresta ou vértice destino de uma aresta PROPER dessas árvores consome memória. Considerando que tinytrees foram concebidas para reduzir o tempo de resposta das consultas via pré-computação, a seguinte questão de pesquisa surge: Será possível construir um algoritmo que identifique quais tinytrees realmente têm de ser materializadas para preservar o baixo tempo de resposta e quais tinytrees podem ser substituídas por uma busca sobre árvores já existentes sem prejuízo significativo no tempo de resposta?

Compartilhamento parcial de séries temporais

A estrutura Tinycubes, bem como as outras estruturas derivadas do Nanocubes, possibilitam o compartilhamento de vértices. No entanto, quando dados temporais não são indexados como dados categóricos e sim armazenados em Contents, não são criados vértices da estrutura para armazenar esses dados e, portanto, o Tinycubes não oferece mecanismos para possibilitar o compartilhamento desses valores. Por exemplo, considere que desde

um momento no tempo t_0 todos os dados recebidos referenciavam apenas um local L_0. Imagine agora que à partir de um momento no tempo t_1 ($t_1 > t_0$) dados referenciando um local L_1 ($L_0 \neq L_1$) passaram a ser recebidos. Neste cenário, um novo vértice terminal teria de ser criado para agregar o conteúdo associado a L_0 e a L_1. Assumindo que os dados contendo informação de tempo são inseridos na estrutura em ordem cronológica, este novo vértice terminal teria dados anteriores a t_1 que não seriam mais modificados. Porém, dado que não é possível compartilhar dados que não sejam diretamente apontados por arestas, é necessário duplicar todos os dados entre t_0 e t_1 no novo vértice terminal, o que pode ocasionar um grande consumo de recursos.

Neste cenário, seria interessante desenvolver uma estrutura de dados para ser utilizada em Containers do Tinycubes que, além de possibilitar rápida recuperação de segmentos de séries temporais, também permitisse o compartilhamento de partes dessas séries entre diferentes instâncias dessa estrutura.

Implementação competitiva de variantes do Nanocubes

O esquema de terminais definido no Tinycubes oferece grande flexibilidade de utilização quando comparado com as soluções existentes. Neste trabalho foram implementados, como prova de conceito, Containers que ofereciam as mesmas funcionalidades extração de informação apresentadas pelo Nanocubes e a parte armazenável das informações recuperáveis pelo Gaussiancubes. No entanto, não foi feita nenhuma implementação de Container otimizada para que fosse possível avaliar a performance da implementação pelo Tinycubes quando comparada a outras variantes da mesma família. Com base nestes fatos, seria interessante pesquisar implementações otimizadas de terminais com as mesmas capacidades de recuperação das variantes para avaliar se um estrutura flexível como o Tinycubes pode substituir soluções especializadas sem perda de eficiência.

Remoção sem necessidade de preservação dos dados inseridos

Para descartar valores armazenados, o algoritmo de remoção do Tinycubes necessita dos mesmos dados que foram responsáveis pelo armazenamento desses valores. Por conta dessa característica, é necessário armazenar os records utilizados na inserção para uma eventual remoção posterior. Se por um lado, esta característica permite a remoção individual de "records" independentemente a qualquer momento, por outro lado exige a manutenção dos dados inseridos na estrutura. Uma questão de pesquisa seria desenvolver um algoritmo de remoção que seja capaz de remover valores armazenados baseados em antiguidade, seja

referente ao tempo em que o valor está armazenado, seja deduzida à partir de algum valor temporal já armazenado.

Utilização de multiprocessamento

Já faz algum tempo que os processadores comerciais possuem capacidades de multiprocessamento. Um processador moderno normalmente possui diversos núcleos (*cores*) de processamento, cada qual com capacidade de executar instruções paralelamente. Os algoritmos implementados atualmente no Tinycubes utilizam apenas um *core*. O desenvolvimento de algoritmos que possam aproveitar eventuais *cores* ociosos pode ser tornar uma questão de pesquisa.

Clusterização

Qualquer sistema computacional possui recursos limitados, seja devido a restrições financeiras, seja devido às limitações de tecnologia. Estruturas de dados como o Tinycubes, que armazenam os seus dados de trabalho inteiramente na memória para reduzir o tempo de resposta, sempre podem sofrer restrições na sua capacidade de prover respostas. Como forma de superar essas limitações de hardware, pode-se pesquisar formas de utilizar os recursos de outros computadores interligados via rede de dados de alta velocidade.

Prova formal da correção dos algoritmos

Os algoritmos desenvolvidos para o Tinycubes foram inspecionados e submetidos a testes de estresse com datasets de diversos tamanhos. No entanto, não foi construída nenhuma prova formal de que os algoritmos realizam as tarefas desejadas sem que ocorram efeitos indesejáveis ao longo do tempo. Uma possível questão de pesquisa é desenvolver provas formais da correção dos algoritmos do Tinycubes.

Referências

[1] Agarwal, S.; Mozafari, B.; Panda, A.; Milner, H.; Madden, S.; Stoica, I. BlinkDB: Queries with bounded errors and bounded response times on very large data. In *Proceedings of the 8th ACM European Conference on Computer Systems, EuroSys 2013* (2013), {ACM} Press, pp. 29–42.

[2] Andrei, N. Kolmogorov. Foundations of the Theory of Probability, 1950.

[3] Andrienko, N.; Gennady, A. Exploratory analysis of spatial and temporal data: A systematic approach, 2006.

[4] Atzori, L.; Iera, A.; Morabito, G. The Internet of Things: A survey. *Computer Networks 54*, 15 (oct 2010), 2787–2805.

[5] Bertin, J. Semiology of graphics: diagrams, networks, maps.

[6] Beyer, K.; Ramakrishnan, R. Bottom-up computation of sparse and Iceberg CUBE. Association for Computing Machinery (ACM), pp. 359–370.

[7] Brehmer, M.; Lee, B.; Bach, B.; Riche, N. H.; Munzner, T. Timelines Revisited: A Design Space and Considerations for Expressive Storytelling. *IEEE TRANSACTIONS ON VISUALIZATION AND COMPUTER GRAPHICS XX*, 1.

[8] Chakrabarti, K.; Garofalakis, M.; Rastogi, R.; Shim, K. Approximate query processing using wavelets. *Proceedings of the 26th International Conference on Very Large Data Bases, VLDB'00 10*, 2-3 (2000), 111–122.

[9] Chaudhuri, S.; Dayal, U. An Overview of Data Warehousing and OLAP Technology. *SIGMOD Record (ACM Special Interest Group on Management of Data) 26*, 1 (1997), 65–74.

[10] Cho, E.; Myers, S. A.; Leskovec, J. Friendship and mobility: User movement in location-based social networks. In *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining* (New York, New York, USA, 2011), ACM Press, pp. 1082–1090.

[11] Chrisman, N. R.; Chrisman, N. R. Fundamental principles of geographic information systems. Auto-Carto 8. *ASPRS & ACSM* (1987).

[12] Cochran, W. G. Sampling Techniques, 1977.

[13] Cuzzocrea, A. Warehousing and protecting big data: State-of-the-art-analysis, methodologies, future challenges. In *ACM International Conference Proceeding Series* (mar 2016), vol. 22-23-Marc, Association for Computing Machinery.

[14] Dahiya, N.; Bhatnagar, V.; Singh, M. Efficient Materialized View Selection for Multi-Dimensional Data Cube Models. *International Journal of Information Retrieval Research 6*, 3 (jul 2016), 52–74.

[15] Dalgleish, D. *Excel 2007 {PivotTables} Recipes*. Apress, 2007.

[16] Deshpande, P. M.; Agarwal, S.; Naughton, J. F.; Ramakrishnan, R. Computation of multidimensional aggregates. Tech. rep., University of Wisconsin-Madison Department of Computer Sciences, 1997.

[17] Diaconis, P.; Bollobas, B. Modern Graph Theory. *Journal of the American Statistical Association* (2000).

[18] Doraiswamy, H.; Freire, J.; Lage, M.; Miranda, F.; Silva, C. Spatio-Temporal Urban Data Analysis: A Visual Analytics Perspective. *IEEE Computer Graphics and Applications 38*, 5 (2018), 26–35.

[19] Du Mouza, C.; Litwin, W.; Rigaux, P. Large-scale indexing of spatial data in distributed repositories: The SD-Rtree. *VLDB Journal 18*, 4 (2009), 933–958.

[20] El-Hindi, M.; Zhao, Z.; Binnig, C.; Kraska, T. VisTrees: Fast indexes for interactive data exploration. In *HILDA 2016 - Proceedings of the Workshop on Human-In-the-Loop Data Analytics* (2016), {ACM} Press.

[21] Etemadpour, R.; Linsen, L.; Paiva, J. G.; Crick, C.; Forbes, A. G. Choosing visualization techniques for multidimensional data projection tasks: A guideline with examples. *Communications in Computer and Information Science 598* (feb 2016), 166–186.

[22] Ferreira, N.; Lage, M.; Doraiswamy, H.; Vo, H.; Wilson, L.; Werner, H.; Park, M.; Silva, C. Urbane: A 3D framework to support data driven decision making in urban development. In *2015 IEEE Conference on Visual Analytics Science and Technology, VAST 2015 - Proceedings* (2015), IEEE, pp. 97–104.

[23] Finkel, R. A.; Bentley, J. L. Quad trees a data structure for retrieval on composite keys. *Acta Informatica 4*, 1 (mar 1974), 1–9.

[24] Friendly, M. Milestones in the history of thematic cartography, statistical graphics, and data visualization, 2009.

[25] Ge, M.; Bangui, H.; Buhnova, B. Big Data for Internet of Things: A Survey. *Future Generation Computer Systems 87* (oct 2018), 601–614.

[26] Govindaraju, N.; Gray, J.; Kumar, R.; Manocha, D. GPUTeraSort: High performance graphics co-processor sorting for large database management. In *Proceedings of the ACM SIGMOD International Conference on Management of Data* (2006), pp. 325–336.

[27] Gray, J.; Chaudhuri, S.; Bosworth, A.; Layman, A.; Reichart, D.; Venkatrao, M.; Pellow, F.; Pirahesh, H. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. In *Data Mining and Knowledge Discovery* (1997), vol. 1, pp. 29–53.

[28] Haerder, T.; Reuter, A. Principles of transaction-oriented database recovery. Tech. Rep. 4, 1983.

[29] Harinarayan, V.; Rajaraman, A.; Ullman, J. D. Implementing Data Cubes Efficiently. In *SIGMOD Record (ACM Special Interest Group on Management of Data)* (1996), vol. 25, pp. 205–216.

[30] Heine, F.; Rohde, M. Popup-Cubing: An algorithm to efficiently use iceberg cubes in data streams. In *BDCAT 2017 - Proceedings of the 4th IEEE/ACM International Conference on Big Data Computing, Applications and Technologies* (dec 2017), Association for Computing Machinery, Inc, pp. 11–20.

[31] Ilyas, I. F.; Beskales, G.; Soliman, M. A. A Survey of Top-k Query Processing Techniques in Relational Database Systems. *ACM Comput. Surv. 40*, 4 (2008).

[32] Jesus, P.; Baquero, C.; Almeida, P. S. A Survey of Distributed Data Aggregation Algorithms, jan 2015.

[33] Krueger, J.; Grund, M.; Jaeckel, I.; Zeier, a.; Plattner, H. Applicability of GPU Computing for Efficient Merge in In-Memory Databases. Tech. rep., 2011.

[34] Lemieux, V. L.; Gormly, B.; Rowledge, L. Meeting Big Data challenges with visual analytics: The role of records management. *Records Management Journal 24*, 2 (jul 2014), 122–141.

[35] Li, M.; Choudhury, F.; Bao, Z.; Samet, H.; Sellis, T. ConcaveCubes: Supporting Cluster-based Geographical Visualization in Large Data Scale. *Computer Graphics Forum 37*, 3 (jun 2018), 217–228.

[36] Li, X.; Hamilton, H. J.; Karimi, K.; Geng, L. The multi-tree cubing algorithm for computing iceberg cubes. *Journal of Intelligent Information Systems 33*, 2 (oct 2009), 179–208.

[37] Lin, K. I.; Jagadish, H. V.; Faloutsos, C. The TV-tree: An index structure for high-dimensional data. *The VLDB Journal 3*, 4 (oct 1994), 517–542.

[38] Lins, L.; Klosowski, J. T.; Scheidegger, C. Nanocubes for real-time exploration of spatiotemporal datasets. *IEEE Transactions on Visualization and Computer Graphics 19*, 12 (2013), 2456–2465.

[39] Liu, C.; Wu, C.; Shao, H.; Yuan, X. SmartCube: An Adaptive Data Management Architecture for the Real-Time Visualization of Spatiotemporal Datasets. *IEEE Transactions on Visualization and Computer Graphics 26*, 1 (jan 2020), 790–799.

[40] Liu, S.; Cui, W.; Wu, Y.; Liu, M. A survey on information visualization: recent advances and challenges. *Visual Computer 30*, 12 (2014), 1373–1393.

[41] Liu, S.; Maljovec, D.; Wang, B.; Bremer, P. T.; Pascucci, V. Visualizing High-Dimensional Data: Advances in the Past Decade. *IEEE Transactions on Visualization and Computer Graphics 23*, 3 (mar 2017), 1249–1268.

[42] Liu, Z.; Heer, J. The effects of interactive latency on exploratory visual analysis. *IEEE Transactions on Visualization and Computer Graphics 20*, 12 (2014), 2122–2131.

[43] Liu, Z.; Jiang, B.; Heer, J. ImMens: Real-time visual querying of big data. *Computer Graphics Forum 32*, 3 PART4 (2013), 421–430.

[44] Lloyd, E. K.; Bollobas, B. Graph Theory: An Introductory Course. *The Mathematical Gazette* (1980).

[45] Malinowski, E.; Zimányi, E. Logical representation of a conceptual model for spatial data warehouses. *GeoInformatica 11*, 4 (dec 2007), 431–457.

[46] McNeill, G.; Hale, S. A. Generating Tile Maps. *Computer Graphics Forum* (2017).

[47] Mei, H.; Ma, Y.; Wei, Y.; Chen, W. The design space of construction tools for information visualization: A survey, feb 2018.

[48] Meyer, P. L. *Probabilidade: aplicações à Estatística.* Ao Livro Técnico S.A., Rio de Janeiro, 1969.

[49] Miller, G. A. The magical number seven, plus or minus two: some limits on our capacity for processing information. *Psychological Review 63*, 2 (1956), 81–97.

[50] Miranda, F.; Lage, M.; . . . , H. D. C. G.; 2018, U. Time lattice: A data structure for the interactive visual analysis of large time series. *Wiley Online Library.*

[51] Miranda, F.; Lins, L.; Kolowski, J.; Silva, C. TopKube: A Rank-Aware Data Cube for Real-Time Exploration of Spatiotemporal Data. *IEEE Transactions on Visualization and Computer Graphics Volume: PP* (2017).

[52] Mitra, S.; Khandelwal, P.; Pallickara, S.; Pallickara, S. L. STASH: Fast Hierarchical Aggregation Queries for Effective Visual Spatiotemporal Explorations. In *Proceedings - IEEE International Conference on Cluster Computing, ICCC* (2019), vol. 2019-Septe, IEEE.

[53] Moritz, D.; Howe, B.; Heer, J. FalCon: Balancing interactive latency and resolution sensitivity for scalable linked visualizations. In *Conference on Human Factors in Computing Systems - Proceedings* (may 2019), Association for Computing Machinery.

[54] Olken, F.; Rotem, D. Sampling from spatial databases. *Statistics and Computing 5*, 1 (mar 1995), 43–57.

[55] Ordonez, C.; Chen, Z.; García-García, J. Interactive exploration and visualization of OLAP cubes. In *International Conference on Information and Knowledge Management, Proceedings* (2011), {ACM} Press, pp. 83–87.

[56] Pahins, C. A. L.; Stephens, S. A.; Scheidegger, C.; Comba, J. L. D. Hashedcubes: Simple, Low Memory, Real-Time Visual Exploration of Big Data. *IEEE Transctions on Visualization and Computer Graphics 23* (2017).

[57] Power, D. J.; Sharda, R. Model-driven decision support systems: Concepts and research directions. *Decision Support Systems 43*, 3 (apr 2007), 1044–1061.

[58] Rivest, S.; Bédard, Y.; Proulx, M. J.; Nadeau, M.; Hubert, F.; Pastor, J. SOLAP technology: Merging business intelligence with geospatial technology for interactive spatio-temporal exploration and analysis of data. *ISPRS Journal of Photogrammetry and Remote Sensing 60*, 1 (2005), 17–33.

[59] Sample, J. T.; Ioup, E. *Tile-Based Geospatial Information Systems*. 2010.

[60] Shao, Z.; Han, J.; Xin, D. MM-cubing: Computing iceberg cubes by factorizing the lattice space. In *Proceedings of the International Conference on Scientific and Statistical Database Management, SSDBM* (2004), vol. 16, pp. 213–222.

[61] Sharda, R.; Barr, S. H.; McDonnell, J. C. Decision Support System Effectiveness: A Review and an Empirical Test. *Management Science 34*, 2 (feb 1988), 139–159.

[62] Shim, J. P.; Warkentin, M.; Courtney, J. F.; Power, D. J.; Sharda, R.; Carlsson, C. Past, present, and future of decision support technology. *Decision Support Systems 33*, 2 (2002), 111–126.

[63] Shneiderman, B. Eyes have it: a task by data type taxonomy for information visualizations. In *IEEE Symposium on Visual Languages, Proceedings* (1996), pp. 336–343.

[64] Shukla, A.; Deshpande, P. M.; Naughton, J. F. Materialized view selection for multi-cube data models. In *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* (2000), vol. 1777, Springer Verlag, pp. 269–284.

[65] Siddiqui, T.; Kim, A.; Lee, J.; Karahalios, K.; Parameswaran, A. Effortless data exploration with zenvisage: An expressive and interactive visual analytics system. In *Proceedings of the VLDB Endowment* (2016), vol. 10, Association for Computing Machinery, pp. 457–468.

[66] Silva, B. N.; Diyan, M.; Han, K. Big Data Analytics. In *SpringerBriefs in Computer Science*. 2019, pp. 13–30.

[67] Silva, J. D.; Vera, A. S. C.; De Oliveira, A. G.; Fidalgo, R. D. N.; Salgado, A. C.; Times, V. C. SBBD 2007 XXII Simpósio Brasileiro de Banco de Dados Querying Geographical Data Warehouses with GeoMDQL. *Simpósio Brasileiro de Banco de Dados XXII* (2007), 223–237.

[68] Siqueira, T. L. L.; de Aguiar Ciferri, C. D.; Times, V. C.; Ciferri, R. R. The SB-index and the HSB-index: Efficient indices for spatial data warehouses. *GeoInformatica 16*, 1 (jan 2012), 165–205.

[69] Tufte, E. R. *The visual display of quantitative information*, vol. 8. Graphics Press, Cheshire, CT, USA, 1988.

[70] Voss, A.; Hernandez, V.; Voss, H.; Scheider, S. Interactive Visual Exploration of Multidimensional Data: Requirements for CommonGIS with OLAP. In *Proceedings of the Database and Expert Systems Applications, 15th International Workshop* (Washington, DC, USA, 2004), DEXA '04, IEEE Computer Society, pp. 883–887.

[71] Wang, A. Techniques for Accelerating Aggregated Range Queries on Large Multidimensional Datasets in Interactive Visual Exploration Item Type text; Electronic Dissertation. Tech. rep., 2019.

[72] Wang, Z.; Cashman, D.; Li, M.; Li, J.; Berger, M.; Levine, J. A.; Chang, R.; Scheidegger, C. NeuralCubes: Deep Representations for Visual Data Exploration.

[73] Wang, Z.; Ferreira, N.; Wei, Y.; Bhaskar, A. S.; Scheidegger, C. E. Gaussian Cubes: Real-Time Modeling for Visual Exploration of Large Multidimensional Datasets. *IEEE Transactions on Visualization and Computer Graphics 23*, 1 (2017), 681–690.

[74] Wu, E.; Psallidas, F.; Miao, Z.; Zhang, H.; Rettig, L.; Wu, Y.; Sellam, T. Combining design and performance in a data visualization management system. In *CIDR 2017 - 8th Biennial Conference on Innovative Data Systems Research* (2017).

[75] Xin, D.; Han, J.; Li, X.; Shao, Z.; Wah, B. W. Computing iceberg cubes by top-down and bottom-up integration: The starcubing approach. *IEEE Transactions on Knowledge and Data Engineering 19*, 1 (jan 2007), 111–126.

[76] Xin, D.; Han, J.; Li, X.; Wah, B. W. Star-cubing: Computing iceberg cubes by top-down and bottom-up integration. In *Proceedings - 29th International Conference on Very Large Data Bases, VLDB 2003* (2003).

[77] Yu, Y.; Gunda, P. K.; Isard, M. Distributed aggregation for data-parallel computing: Interfaces and implementations. In *SOSP'09 - Proceedings of the 22nd ACM SIGOPS Symposium on Operating Systems Principles* (2009), pp. 247–260.

[78] Zhang, J.; You, S.; Gruenwald, L. High-performance online spatial and temporal aggregations on multi-core {CPUs} and many-core {GPUs}. In *Proceedings of the fifteenth international workshop on Data warehousing and {OLAP} - {DOLAP} {\textquotesingle}12* (2012), {ACM} Press.

[79] Zhang, J.; You, S.; Gruenwald, L. Large-scale spatial data processing on {GPUs} and {GPU}-accelerated clusters. *{SIGSPATIAL} Special 6*, 3 (2015), 27–34.