Estratégias de Particionamento de Blocos para o Grid File

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Fernando Rodrigues de Almeida Júnior e aprovada pela Banca Examinadora.

Fortaleza, 17 de janeiro de 2002.

Angelo Roncalli Alencar Brayner (UNIFOR)
(Orientador)

Dissertação apresentada ao Mestrado em Ciência da Computação, UFC, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

Mestrado em Ciência da Computação

Universidade Federal do Ceará

Estratégias de Particionamento de Blocos para o Grid File

Fernando Rodrigues de Almeida Júnior

Janeiro de 2002

Banca Examinadora:

- Angelo Roncalli Alencar Brayner (UNIFOR)
- Creto Augusto Ponte Vidal (UFC)
- Geovane Cayres Magalhães (UNICAMP)

Dedicatória

Dedico este trabalho a todos aqueles que sabem ver o mundo com olhos diferentes, não ficando passivos aos fatos que lhes são apresentados, mas buscando sempre mais e melhores explicações para as coisas que acontecem ao seu redor: A todos os cientistas, não só de profissão, mas de alma.

Agradecimentos

Agradeço, primeiramente, a Deus e a minha família, principalmente a meus pais, pela força que me deram nesta caminhada para concluir este trabalho ao longo destes anos.

Em segundo lugar, gostaria de agradecer também à Érika Vieira que me incentivou, me deu atenção e carinho durante muitos e muitos dias que me senti desamparado.

Devo um enorme agradecimento ao Prof. Angelo Brayner, que me orientou e me deu total apoio. Ao Prof. Creto Vidal, que cooperou co-orientando o trabalho e ao Prof. Joaquim Bento, que me deu boas idéias durante a fase de pesquisa por novas estratégias de busca. Ao Prof. Geovane Cayres, que fez uma excelente revisão e colaborou com preciosas críticas para o refinamento do documento final.

Não posso esquecer de agradecer ao CNPq (Conselho Nacional de Desenvolvimento Científico e Tecnológico), que deu suporte financeiro, através de uma bolsa, para a realização deste projeto.

Obrigado a todos os colegas do mestrado e amigos em geral, ao pessoal da Peleja que me animou e divertiu, fazendo mais suave o longo caminho até a conclusão deste trabalho.

Por fim, a todos aqueles que ajudaram direta ou indiretamente para que este trabalho fosse concluído com êxito, e que eu não tenha citado aqui.

A todos, meu muito obrigado!

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

1.1. Motivação

Dados espaciais são tipos de dados multivalorados e, assim sendo, precisam ser acessados através de diversos atributos simultaneamente, isto é, devem ser tratados multidimensionalmente. Como exemplo de dados espaciais, pode-se citar os pontos representativos das cidades de uma região em determinadas aplicações geográficas, que necessitam ser indexados utilizando-se dois parâmetros, x e y, respectivamente, os atributos longitude e latitude em relação ao globo terrestre.

Dados espaciais devem ser armazenados e indexados por estruturas que suportem os chamados Métodos de Acesso Espacial (ver seção 2 do artigo [40]), pois as estruturas tradicionais de indexação, projetadas para dados simples, não garantem um desempenho mínimo necessário ao uso de dados espaciais.

Um exemplo do uso de múltiplas chaves de busca pode ser observado em um *Sistema de Informação Geográfica* (SIG) que lida com fotos enviadas por satélites. Suponha que se deseja obter, como resultado de uma consulta, a data da ocorrência da última queimada nas proximidades de um determinado ponto de longitude x e latitude y e uma imagem da devastação ocorrida na região. Para responder a essa consulta, deverão ser utilizados os atributos x e y para recuperar os registros do local desejado e uma imagem do mesmo. Nesse caso, será dispendioso, do ponto de vista computacional, o uso de B-Trees[32].

Estruturas convencionais como B-Trees utilizam um único atributo como chave de busca e, assim, as consultas sobre múltiplos atributos deveriam ser convertidas em várias outras consultas sobre cada atributo-chave, requerendo bastante tempo de processamento para que sejam feitas as devidas conversões. No caso de estruturas como Arquivos Invertidos, que são extensões de estruturas originalmente reforçadas para acesso a registros por chave de busca única, o problema da adaptação aos arquivos altamente dinâmicos (com muitas inserções e remoções) não é tratado adequadamente, apesar dessa estrutura suportar o acesso por múltiplas chaves.

Uma das estruturas propostas com a finalidade de suportar o armazenamento e a indexação de dados espaciais é o *Grid File* (Arquivo Grade)[3], que são estruturas de arquivo do tipo *hashing*. Os Arquivos Grades funcionam através das correspondências das *células* de uma grade com os seus respectivos *buckets* por meio de ponteiros. Um *bucket* é uma unidade de armazenamento físico no disco onde são mantidos os registros. O conteúdo de um *bucket* pode ser alocado completamente em memória principal. *Células*, ou *blocos*, são as unidades de alocação lógicas que representam cada quadrante da grade de um *Grid File*. Um *bucket* é dito *simples* quando ele é referenciado por uma única célula da *grade*, ou seja, a região do *bucket* corresponde a uma célula. Um *bucket* é *composto* quando mais de uma célula da *grade* utiliza-o para armazenar os seus registros, isto é, a sua região é composta por duas ou mais células.

Dizemos que um *bucket* está sobrecarregado quando sua capacidade máxima de registros é atingida. Quando a inserção de um novo registro é endereçada a uma célula correspondente a um *bucket* simples sobrecarregado, ocorre o *bucket overflow*, quando deve ocorrer a divisão desta célula em uma certa dimensão, para que os novos *buckets* recém-criados possam suportar uma maior quantidade de registros naquela região da *grade*.

Uma abordagem para realizar essa divisão das células da *grade* é apontada por [5] como sendo a divisão de células exatamente no ponto médio de uma certa dimensão escolhida, mas essa política de divisão não se mostra eficiente quando os registros ficam agrupados

em cantos da célula, pois dessa maneira, após o particionamento, os registros continuariam ocupando um só *bucket* que também teria atingido sua capacidade máxima.

Grid Files são utilizados em aplicações que necessitam acessar registros através de múltiplas chaves de busca, como aplicações de Projeto Assistido por Computador (CAD) e os SIGs. Tais aplicações avançadas da tecnologia de Bancos de Dados utilizam vários atributos para identificar e/ou manipular os seus objetos.

1.2. Objetivos da Dissertação

O presente trabalho tem como objetivo básico desenvolver e implementar políticas de divisão de células (e, conseqüentemente, de grade) para a estrutura de arquivo conhecida por *Grid File*. Adicionalmente, será apresentada uma revisão bibliográfica do estado da arte das pesquisas relacionadas aos *Grid Files* e estruturas para armazenar dados espaciais no terceiro capítulo dessa dissertação.

As políticas propostas melhoram o desempenho geral do *Grid File* através do modo como o mesmo fará a divisão de células, quando houver o *overflow* de um *bucket*, visto que, geralmente, há constantes inclusões de novos dados em aplicações como os Sistemas de Informação Geográfica (SIG), que tratam constantemente com objetos de representações geométricas, como figuras, mapas, fotos de satélites, maquetes etc.

A partir das políticas propostas, foi implementado um protótipo para armazenamento de dados multidimensionais. Esse protótipo utiliza a abordagem de *Grid Files* e pode servir de mecanismo de armazenamento para um SIG.

Uma implementação efetiva das políticas propostas poderá ser realizada no SGBD Informix®, na forma de DataBlade®. Os módulos DataBlade do Informix funcionam como extensões do banco de dados, pois permitem que as funcionalidades do Informix sejam incrementadas de uma maneira nativa, através de programas em C ou em Java, executados a partir dos módulos DataBlade.

1.3. Organização da Dissertação

Esta dissertação está estruturada da seguinte forma: no Capítulo 2 é apresentada a definição do problema tratado. Inicialmente, a estrutura dos *Grid Files* é conceituada. Em seguida, os problemas das operações de divisão (splitting) e união (merge) de células em *Grid Files* é descrito e analisado.

No Capítulo 3, é apresentada uma revisão bibliográfica das propostas de estruturas de arquivos para armazenamento e acesso por múltiplas chaves de busca, seguida da justificativa da escolha de *Grid Files* para esse fim em SIGs.

Os algoritmos propostos nesta dissertação para a otimização da operação de divisão de células no *Grid File* serão apresentados no Capítulo 4.

No Capítulo 5, é apresentada uma descrição do protótipo implementado em Java para simular e mensurar o funcionamento de um sistema de armazenamento utilizando *Grid Files*, e são mostrados os resultados dos testes realizados no protótipo implementado. Finalmente, no Capítulo 6, são apresentados as contribuições deste trabalho, alguns resultados e as conclusões. São também discutidos possíveis trabalhos futuros que podem dar continuidade a este trabalho.

Capítulo 2

Grid Files: Taxonomia e Operações

2.1. Introdução

Neste capítulo, são abordados mais detalhadamente o funcionamento da estrutura de *Grid Files* e a nomenclatura usada para suas diversas sub-partes. Na Seção 2.2, é apresentada a estrutura de *Grid Files* de uma maneira mais genérica, descrevendo-se seus vários componentes e mostrando-se um exemplo de seu uso prático. Na Seção 2.3, aborda-se o uso de *Grid Files* em aplicações georeferenciadas, mostrando-se suas qualidades e suas desvantagens quando comparado a outras propostas de estruturas de arquivos. Finalmente, na Seção 2.4, explica-se como funcionam os processos de divisão e união de células em *Grid Files*, encerrando assim este capítulo.

2.2. *Grid Files*

Um *Grid File* representa uma estrutura para arquivos de dados que garante acesso por múltiplas chaves de busca aos registros. Por esse motivo, os *Grid Files* são estruturas eficientes para o armazenamento de dados espaciais.

O *Grid File* é uma estrutura de arquivos projetada para gerenciar blocos de armazenamento em disco de tamanho fixo chamados *buckets*. Cada *bucket* tem capacidade de conter uma certa quantidade de registros, onde cada registro contém propriedades(por exemplo,

tipo de relevo ou tipo de solo) de um ponto do domínio. A capacidade de armazenamento de registros em cada *bucket* será igual ao tamanho da memória principal(tamanho do *bucket*) dividido pelo tamanho de cada registro a ser armazenado no *Grid File*.

A idéia é particionar uma região k-dimensional em uma grade também k-dimensional. Por exemplo, uma região bidimensional deve ser particionada em uma grade bidimensional como ilustrado na Figura 2.1.

A estrutura usada para organizar o conjunto de *buckets* é chamada de *grid directory*. É através dessa estrutura que se definem as correspondências entre as *células* da grade e os *buckets* no disco.

O *grid directory* consiste de dois componentes. Para fazer a ligação de cada célula da grade com os valores reais dos atributos indexados entra em cena a primeira componente do *grid directory*, as *escalas lineares,* k vetores unidimensionais que definem as partições do domínio do atributo correspondente. O segundo componente é um vetor de k dimensões, onde k é o número de atributos que serão usados como chave de indexação. Cada elemento e[i][j]...[k] deste vetor é um ponteiro para um *bucket*. Esse *bucket* contém os registros correspondentes à área do elemento e[i][j]...[k] dentro da grade, chamado de *célula*. Este vetor é conhecido como *grid array*.

A Figura 2.1 mostra um exemplo de estrutura de *Grid File* bidimensional. As ligações entre as células da grade e os seus respectivos *buckets* são ilustradas como setas. As escalas lineares são constituídas pelos eixos x e y, os quais são divididos respectivamente nos pontos *x0, x1, x2, x3* e *y0, y1, y2*. Os registros de cada célula do *grid directory* são mapeados em *buckets* através das escalas lineares e do *grid array*. Por exemplo, os registros contidos na célula G[*x0_x1, y1_y2*] são armazenados fisicamente no *bucket* b1, enquanto que os registros pertencentes a célula G[*x2_x3, y0_y1*] são alocados no *bucket* b3 juntamente com os registros da célula G[*x2_x3, y1_y2*].

Como já mencionado no início desta seção, os *Grid Files* armazenam e indexam de forma eficiente dados espaciais. Essa eficiência decorre, basicamente, de duas propriedades garantidas pela estrutura de *Grid Files*:

Figura 2.1. *Layout de um Grid File com 2 dimensões*

1. O Princípio dos dois acessos ao disco: Uma consulta em um *Grid File* recupera um registro em, no máximo, dois acessos ao disco; e

2. A eficiência de consultas por faixas: A estrutura de *Grid Files* garante que os registros que estão próximos no domínio de qualquer atributo são armazenados no mesmo *bucket* ou em *buckets* adjacentes.

A primeira propriedade deriva da maneira como funciona a estrutura dos *Grid Files*, através do *grid directory*. Para recuperar um determinado registro, são passados como parâmetros os valores de cada atributo usado como chave de busca. Então esses valores são transformados para coordenadas do *grid array* através da busca pelos índices correspondentes das escalas lineares, que já se encontram em memória principal, por serem vetores de tamanho reduzido. O *grid array* é então recuperado do disco, o que constitui o primeiro acesso, e então com a comparação dos devidos valores dos índices de cada atributo chave, se encontra a célula que contém o registro a ser recuperado. O conteúdo da célula é um ponteiro para um *bucket* no disco que contém os registros equivalentes à

célula do ponto procurado. Neste momento se faz o segundo acesso ao disco para colocar o *bucket* em memória principal.

A segunda propriedade, por sua vez, é garantida pelo princípio da localidade de registros em estruturas que utilizam Técnicas da Computação de Endereço ou *Hashing*, onde os limites das regiões de armazenamento(blocos ou células) não são definidos por meio dos pontos(registros) armazenados na estrutura, e sim através de todo o domínio dos atributos-chave do arquivo.

Os *Grid Files* apresentam dois tipos de operações: operações sobre objetos em disco e operações sobre o *grid directory*. As operações sobre objetos são inserção, remoção e atualização (alteração), além de consulta.

Vale ressaltar que as partições podem ser modificadas em resposta às operações de inserção e remoção. Por exemplo, uma partição unidimensional pode ser modificada dividindo-se um de seus intervalos em dois ou unindo-se dois intervalos adjacentes em um só.

As operações que podem ser executadas sobre o *grid directory* são: Acesso direto, próximo em cada direção, união de duas partições e divisão de uma partição.

2.3. A Utilização de Grid Files para Armazenar e Indexar Dados Georeferenciados

Os *Grid Files* são apropriados, por vários aspectos já citados anteriormente, para armazenar dados multivalorados em Sistemas de Informação Geográfica, pois eles preenchem vários requisitos básicos para tal tarefa, além de oferecer capacidades extras de estruturas de dados que facilitam a manutenção das informações de uma maneira mais natural que outros tipos de estruturas, como as árvores.

A estrutura dos *Grid Files* é capaz de armazenar grandes quantidades de dados organizados através de múltiplos índices, podendo usar até dez (10) atributos como chaves de indexação. Esta capacidade contribui para o seu uso em sistemas de grande porte, como

a maioria dos SIG's utilizados comercialmente, onde algumas propriedades, tais como a localização (latitude, longitude, altitude) de certos objetos geográficos, como regiões, cidades, bairros e ruas, são utilizadas para localizar e manter registros/objetos no banco de dados.

2.4. Divisão e União de Células em *Grid Files*

Uma divisão de uma partição da grade é disparada quando se tenta inserir um registro em um *bucket* simples cuja capacidade máxima de armazenamento já foi atingida. Nesse caso deve-se escolher a dimensão (o eixo para o qual o hiperplano de particionamento é ortogonal) e a localização (o ponto no qual a escala linear é particionada) desta divisão. No caso da mesma situação acontecer com um *bucket* composto, não é necessário o particionamento da grade, visto que basta alocar-se um novo *bucket* no disco para separar os registros por uma ou mais células pertencentes a sua região.

Várias políticas de divisão de blocos são aplicáveis ao *Grid File*; elas resultam em diferentes divisões da partição da grade. O implementador, ou até mesmo o usuário de uma implementação do *Grid File* suficientemente genérica, pode escolher uma, entre várias políticas possíveis, na tentativa de otimizar o desempenho da aplicação com base na freqüência de consultas observadas em seu aplicativo.

As políticas de divisão mais simples escolhem a dimensão de acordo com um procedimento fixo, geralmente de maneira cíclica. Uma política de divisão pode favorecer alguns atributos (no sentido de construir uma escala linear de maior resolução) dividindo as dimensões correspondentes mais frequentemente do que as outras. Isto acarreta o efeito de aumentar a precisão das respostas para consultas especificadas parcialmente em que os atributos favorecidos são especificados, mas os outros não são.

A política de divisão proposta em [3] particiona a célula da grade exatamente no ponto médio do intervalo em relação a uma determinada dimensão, o que torna essa política computacionalmente eficiente (O(1)), mas ao mesmo tempo, requer que em muitas situações, ocorra um maior número de particionamentos, devido à inserção de muitos

registros de dados muito próximos dentro de uma mesma grade.

A Figura 2.2 mostra um exemplo de divisão de um *bucket* de dados sobrecarregado seguindo a política de divisão de células tradicional do *Grid File*. Os pontos representam a localização dos registros de dados dentro do *bucket* e as linhas pontilhadas representam os possíveis locais de divisão da grade em cada dimensão.

Figura 2.2. *Política de divisão tradicional do Grid File*

A localização de uma divisão sobre uma escala linear não precisa necessariamente ser escolhida no meio do intervalo da célula. Ela pode ser feita em qualquer ponto da célula, desde que seja estendida por todo o espaço da grade na dimensão em questão. A única restrição é o número de pontos usados para particionar a grade. Se este número for maior do que um, dividindo um intervalo da grade em mais de dois novos intervalos, esta divisão não pode ser estendida para os *buckets* pois, por experimentos práticos demonstrados em [5], verificou-se que a taxa da média de ocupação dos *buckets*, no caso da divisão de cada intervalo por três novos intervalos, cai, dos 70% esperados, para apenas 39%. Isso torna o uso do *Grid File* inviável, do ponto de vista do espaço ocupado em disco por um arquivo desse tipo.

A política de divisão proposta nesta dissertação melhora o desempenho do *Grid File*

em aplicações como os Sistemas de Informação Geográfica. Tais aplicações tratam constantemente com representações geográficas, como mapas, maquetes, medidas topográficas, etc, que necessitam de uma política de divisão que dê suporte às necessidades específicas desse tipo de aplicação.

Quando ocorrem muitas remoções (exclusões) de registros em um *Grid File*, alguns *buckets* passam a ficar com a taxa de ocupação abaixo do limite mínimo fixado, caracterizando o chamado "*underflow*". Nessa situação, o *Grid File* dispara o mecanismo de união de células vizinhas, sempre atendendo à regra de que as células "unidas" (regiões dos *buckets*) tenham um formato retangular(*box-shaped*). Dessa maneira, tenta-se manter um patamar mínimo de ocupação dos *buckets* (geralmente 40%) e a estrutura do *grid directory* continua a ser eficiente, ao mesmo tempo que economiza espaço em disco, descartando a possibilidade de alocação de *buckets* "quase" vazios.

Apesar da importância do processo de união para o funcionamento do *Grid File*, essa função tem se mostrado menos utilizada em implementações de SIGs do que o processo inverso, já citado anteriormente, conhecido por divisão, devido a um maior número de inclusões de registros do que exclusões.

Capítulo 3

Revisão Bibliográfica

3.1. Introdução

Neste capítulo, inicialmente é feita uma análise do estado-da-arte das estruturas e dos sistemas que abordam o tratamento de dados multidimensionais. Começa-se pelos sistemas de nível mais alto e, conseqüentemente, de propósito mais específico, conhecidos como Sistemas de Informação Geográfica (SIGs) mostrando os tipos de dados que são tratados por estes sistemas.

Em seguida, na Seção 3.2.1, é exibida uma arquitetura genérica dividida em camadas e subsistemas para o projeto de um SIG, dando a descrição de cada uma das camadas e dos objetos que compõem a definição do modelo. Na Seção 3.2.2, as fases de construção e implementação de um SIG são discutidas com mais detalhes.

Na Seção 3.3, são analisadas as estruturas de armazenamento de dados multivalorados mais genéricas que podem ser usadas para armazenar objetos, tais como aqueles encontrados em Sistemas de Informação Geográfica e em aplicações do tipo CAD.

Em seguida, na Seção 3.4, é apresentada uma descrição mais específica da estrutura de *Grid Files*. Finaliza-se este capítulo, discutindo-se e analisando-se as abordagens mais importantes para divisão de células em *Grid Files*.

3.2. Sistemas de Informação Geográfica

Estruturas de dados que suportam acesso/indexação através de múltiplas chaves de busca são necessárias para vários tipos de aplicações avançadas de BDs, como, por exemplo, para aplicações que manipulam e gerenciam grandes conjuntos de dados espaciais. Um exemplo clássico de uma aplicação com estas características são os Sistemas de Informação Geográfica (SIG).

Um Sistema de Informação Geográfica (SIG) consiste de um conjunto de ferramentas computacionais usadas para consultar, ordenar e manusear dados estatísticos e gráficos sobre uma determinada área geográfica, de uma maneira uniformizada e prática.

Os dados que podem ser armazenados em um SIG podem ser classificados como:

- Convencionais: classe de dados que consistem de atributos uni-dimensionais (atributos alfa-numéricos e/ou numéricos), por exemplo: NomeCidade, População. Estes dados são armazenados como relações, em BDs relacionais, ou classes, em BDs OO;

- Espaciais: dados que consistem de coordenadas e outras propriedades espaciais, como área, volume, etc, definindo os esquemas dos objetos espaciais dos quais as propriedades correspondentes são armazenadas como relações convencionais. A descrição dos objetos espaciais pode ser obtida usando-se os identificadores de objetos;

- Gráficos: são usados principalmente com o propósito de exibição e geralmente estão na forma de mapas. Além de operações sobre dados gráficos (zoom, pan), este tipo de dado permite operações nas quais estes são relacionados com dados convencionais e/ou espaciais. Pode-se usar a figura de um mapa, por exemplo, para gerar uma descrição não pictórica da imagem, através de um processo conhecido como *redução de dados*[1]. Por este processo, em um SIG, o usuário pode definir uma janela de uma região de um mapa e, por exemplo, realizar uma consulta que deve retornar todas as cidades com mais de 10 mil habitantes que se localizam a até 10 Km da área

[1] *O uso de técnicas de fotointerpretação e de classificação automática é necessário para individualizar os objetos geográficos contidos na imagem.*

selecionada. Conseqüentemente, note que por este processo, o gráfico é relacionado aos seus atributos espaciais.

Enquanto os dados gráficos geralmente são usados para exibição de resultados de consultas feitas ao SIG, os dados espaciais e convencionais, por sua vez, são manipulados pelo processador de consultas do sistema para realizar o processamento desejado.

3.2.1. Camadas e Subsistemas de uma Arquitetura para SIG's

Uma estratégia geralmente empregada para descrever sistemas computacionais consiste em dividi-los funcionalmente em camadas e, ortogonalmente, em subsistemas especializados em determinada tarefa ou em determinado tipo de dados.

Para uma melhor compreensão das estratégias de organização de um sistema que suporta informações georeferenciadas, descreve-se, a seguir, uma arquitetura em camadas para um SIG que claramente separa as funções de armazenamento, manipulação e visualização.

A Figura 3.1, retirada da página 108 de [12], indica as camadas e subsistemas propostos para organizar um SIG nesse trabalho. Em um nível superior ao primeiro nível da figura, e não especificado por esta arquitetura, encontram-se as aplicações de geoprocessamento e os módulos de interface com o usuário.

O primeiro nível divide os problemas de visualização dos problemas de manipulação e corresponde ao nível conceitual (abstrato) do modelo de arquitetura genérica em camadas para um Sistema de Informação Geográfica. O *Subsistema de Visualização* (SV) fornece funções básicas para visualização tanto de objetos tradicionais como de objetos georeferenciados. Por sua vez, o *Subsistema de Manipulação* (SM) oferece funções para definição e manipulação dos objetos a serem tratados pelo sistema.

O segundo nível corresponde ao nível de representação do modelo e oferece separadamente serviços de manipulação de alto nível para atributos convencionais, como textos e dados numéricos, através do *Subsistema de Manipulação Convencional* (SMC), representações matriciais, como mapas de bits (bitmaps), através do *Subsistema de Mani-*

Figura 3.1. *Arquitetura genérica em camadas para um SIG.*

pulação Matricial (SMM) e representações vetoriais, como gráficos construídos através de cálculos de operações aritméticas sobre equações vetoriais, através do *Subsistema de Manipulação Vetorial* (SMV).

O terceiro nível incorpora os subsistemas que fornecem serviços de armazenamento e manipulação elementar para atributos convencionais (*Subsistema de Armazenamento Convencional* - SAC), representações matriciais (*Subsistema de Armazenamento Matricial* - SAM) e representações vetoriais (*Subsistema de Armazenamento Vetorial* - SAV).

Estes subsistemas fazem uso do serviço de armazenamento de páginas físicas disponibilizado pelo *Subsistema de Armazenamento Físico* (SAF), que compõe o quarto e mais baixo nível da arquitetura. É nesse nível que se encontram as estruturas de armazenamento, como os *Grid Files*, que fazem o tratamento dos dados que serão armazenados e recuperados do disco.

3.2.2. Estratégias de Implementação

A implementação de um SIG pode ser realizada utilizando-se duas estratégias distintas, como descrito a seguir:

- **Projetar e implementar um novo sistema de propósito especial.** Este novo sistema suportaria uma linguagem de consulta de propósito especial e mecanismos de acesso, tipos de dados estendidos e um processador gráfico especial para conciliar o armazenamento e a recuperação de dados espaciais e convencionais.

- **Estender um sistema existente para suportar o gerenciamento de dados espaciais.** Esta abordagem incluiria a extensão de um SGBD convencional para prover acesso para dados espaciais e não-espaciais, e para se comunicar com um subsistema gráfico responsável por todas as operações de entrada e saída gráfica. Por exemplo, no caso do SGBD Informix, estas estruturas poderiam ser implementadas na forma de Data Blades.

Dentro da segunda abordagem, existem três principais estratégias de implementação de SIG e elas permitem classificar, de maneira coerente, vários SIG disponíveis comercialmente ou propostos como protótipos. A tabela a seguir, retirada da página 109 de [12], mostra as vantagens e os problemas no uso de cada uma das estratégias.

Estratégia	*Vantagens*	*Problemas*
Dual	SGBD de mercado Independência do SGBD	Controle de integridade Otimização de Consultas
Campos Longos	Controle de integridade	Dependência do SGBD Otimização de Consultas
Extensível	Controle de integridade Otimização de Consultas e métodos de indexação	Dependência do SGBD Falta de Padronização aceita para ling. de consulta

Tabela 3.1. *Estratégias de implementação de SIG's [12].*

A estratégia *dual* baseia-se na utilização de um SGBD relacional para armazenar em forma de tabelas a componente convencional de todos os objetos incorporados, e utiliza-se dos arquivos normais do sistema operacional hospedeiro para armazenar a componente espacial dos objetos geográficos. Dessa maneira, qualquer SGBD relacional pode ser

adotado para dar suporte ao Sistema de Informação Geográfica usando a abordagem dual.

De acordo com a arquitetura em camadas mostrada na Figura 3.1, a estratégia dual determina que os subsistemas de manipulação convencional (SMC) e o de armazenamento convencional (SAC) sejam inteiramente implementados sobre um SGBD relacional, e os subsistemas de manipulação matricial (SMM), de armazenamento matricial (SAM), de manipulação vetorial (SMV) e de armazenamento vetorial (SAV) sejam implementados sobre o sistema de arquivos do sistema operacional utilizado, para a componente espacial das representações, e sobre o SGBDR adotado, para a componente convencional das representações.

Os sistemas disponíveis comercialmente segundo esta estratégia não conseguem encobrir a dualidade para o usuário, pois a mesma é levada até os níveis superiores da arquitetura. Exemplos de sistema que utilizam esta estratégia são o ARC/INFO[34] e o sistema SPRING ([35] e [36]), que faz exceção a regra e mascara a dualidade para as camadas superiores.

A estratégia de *campos longos* baseia-se no uso de SGBDs relacionais com suporte para campos do tipo *BLOB (Binary Large Objects)*, onde se pode armazenar grandes cadeias binárias, como as componentes espaciais dos objetos geográficos.

No que se refere a arquitetura em camadas, pode-se dizer que tanto os subsistemas responsáveis pela manipulação dos atributos convencionais - SMC - e os subsistemas responsáveis pelo armazenamento dos atributos convencionais - SAC - quanto os subsistemas para armazenamento matricial e vetorial - SAM e SAV - passam a ser implementados, em parte, através do SGBDR. Porém, os subsistemas para manipulação matricial e vetorial - SMM e SMV - continuam externos ao SGBD, isto é, continuam como arquivos normais do sistema operacional adotado.

Conceitualmente, um campo longo é definido apenas como uma cadeia binária onde podem ser armazenados quaisquer tipos de dados, tais como gráficos, planilhas, textos ou imagens. Fazendo-se uma abodagem prática, os SGBDs atuais permitem que este tipo

de campo possua um tamanho considerável, podendo-se alocar espaço até da ordem de gigabytes para tais campos.

A grande vantagem da abordagem de campos longos é armazenar todos os objetos de um banco de dados geográficos em um único SGBD, evitando os problemas de gerência de transações e de controle de integridade e de concorrência, que existem na estratégia dual, decorrentes do fato de que a componente espacial das representações fica armazenada externamente ao banco de dados, através do próprio sistema de arquivos do ambiente operacional adotado.

Por outro lado, o SGBD não conhece a semântica dos dados que estão armazenados em campos longos, pois os trata apenas como seqüências binárias e, portanto, não possui mecanismos para um tratamento mais adequado aos grandes volumes de dados. O SYSTEM 9 [17] é um exemplo de sistema implementado segundo esta estratégia.

A estratégia baseada em *mecanismos de extensão* é ainda mais restritiva que a segunda em relação a escolha do SGBD, pois depende de que este disponha de mecanismos que permitam implementar o tratamento das componentes espaciais através de extensões ao seu ambiente. São os chamados SGBD's extensíveis.

Um SGBD extensível permite a definição de novos tipos de objetos através da linguagem de definição e manipulação de dados, isto é, da sua interface com o usuário. Deve-se atentar para o fato de que a especificação de um novo tipo de dado inclui a definição dos atributos dos objetos e dos métodos que atuam sobre estes objetos. Desta maneira, esta facilidade vai muito além de campos longos, pois permite capturar a semântica dos objetos. São exemplos de SGBDs extensíveis: o POSTGRES[37], o Informix (Data Blades), o Oracle (Cartridges) e os sistemas orientados a objetos, como o O_2[38] e o ObjectStore.

3.3. Estruturas de Armazenamento

Existem várias estruturas de dados propostas para suprir os requisitos de uma boa base para armazenar os dados em um SIG, sempre tratando o aspecto da necessidade de multiplicidade dos índices em aplicações que se propõem a armazenar objetos geométricos

(figuras) e/ou geográficos (mapas). Um outro pré-requisito para uma boa estrutura de armazenamento e recuperação de bancos de dados geográficos é o bom desempenho em consultas por campos (regiões) ou pontos específicos do espaço total de busca.

Algumas destas estruturas projetadas são as *Quad-trees* (Point [20] e Region [21]), as *Kd-trees* [22], as *KDB-trees* [6], as *Mkd-trees* (ou Matsuyama's kd-trees) [23], as *Skd-trees* [13], as *R-trees* [2], as *R+-trees* [7], *Corner Stiching* [24], *EXCELL* [8] [25], *PLOP-Hashing* [26], *Quad-CIF-tree* [27] [28], *Locational Keys* [29], *4-D-tree* [30], *Cell-trees* [31] e os *Grid Files* [5].

As Quad-Trees são estruturas do tipo "árvore" que particionam um espaço k-dimensional em células irregulares, onde cada nó representa um ponto e possui 2^k filhos, cada um destes representando uma direção (NE, NW, SW, SE, por exemplo, para k igual a 2). As folhas representam áreas do espaço e apontam para *buckets* no disco, sendo que várias folhas podem apontar para um mesmo *bucket*. A Figura 3.2 representa uma Quad-Tree de uma região de tamanho 100 x 100 contendo seis pontos.

Figura 3.2. *Quad-Tree com duas dimensões e seis pontos (à esquerda) e a sua árvore de representação (à direita)*

No exemplo da Figura 3.2, o ponto P_1 de coordenadas x=55 e y=45 é a raiz da árvore. Seus nós-filhos são, respectivamente, P_5(70,60), P_2(35,80), P_3(10,35) e P_6(60,25). O nó

P_3 é pai de P_4(20,15).

As KD-Trees também particionam um espaço k-dimensional em regiões irregulares, com cada nó representando um ponto e tendo 2 filhos, um para cada sub-espaço resultante do particionamento ocorrido. Geralmente a dimensão para o particionamento é escolhida de maneira cíclica[2]. Cada nó ainda deve indicar em qual dimensão o particionamento do sub-espaço foi feito. Assim como nas Quad-Trees, as folhas representam áreas do espaço e apontam para *buckets* no disco, com várias folhas podendo apontar para um mesmo *bucket*. A Figura 3.3 representa uma KD-Tree de uma região com área igual a 100 x 100 contendo 8 pontos, ou seja, com 8 particionamentos do espaço total. À direita temos a representação do espaço particionado em forma de árvore.

Figura 3.3. *KD-Tree com duas dimensões e oito pontos (à esquerda) e a sua árvore de representação (à direita)*

As R-Trees particionam um espaço k-dimensional R em retângulos usando a técnica de *Retângulo Envolvente Mínimo (REM)*[3] para um determinado conjunto de pontos do espaço em questão. Cada nó representa retângulos de áreas do espaço e cada folha pode conter tanto retângulos (objetos não-pontuais no espaço) como pontos, podendo haver

[2]Por exemplo, para um espaço com k igual a 2 (bidimensional), se um nó tem o particionamento no eixo x, então seus filhos são particionados pelo eixo y, e vice-versa.

[3]*Menor retângulo que contém um determinado conjunto de objetos ou pontos.*

superposição de folhas. Todos os objetos pertencentes a uma folha podem ser representados como entradas do tipo [*r*, *ponteiro*], onde *r* representa um retângulo de R (ou seja, um identificador ou chave de busca) e *ponteiro* contém o endereço da página onde estão armazenados fisicamente os objetos contidos em r. Vale ressaltar que as R-Trees são uma generalização k-dimensional das B-Trees, e, portanto, são balanceadas pela altura.

Figura 3.4. *R-Tree de duas dimensões com 8 REM, 8 objetos pontuais e 9 objetos retangulares*

A Figura 3.4 representa um mapeamento de uma R-Tree constituindo-se de oito retângulos envolventes mínimos [R_1, R_2, R_3, R_4, R_5, R_6, R_7, R_8], contendo oito pontos [p_1, p_2, p_3, p_4, p_5, p_6, p_7, p_8], representando objetos pontuais do espaço, e nove retângulos [r_1, r_2, r_3, r_4, r_5, r_6, r_7, r_8, r_9], representando regiões do espaço.

As estruturas do tipo "árvore" (*trees*), em geral, demandam um maior poder de processamento para realizarem operações de atualização de registros (inserções e remoções, por exemplo).

Nestas estruturas, em muitas situações, uma simples remoção de um registro, usado como nó, dispara todo um processo de reorganização da árvore, sendo, muitas vezes, este processo expandido para os níveis inferiores da árvore, o que ocasiona um *delay* no

desempenho geral do sistema, devido à natureza exponencial da operação.

Por outro lado, as estruturas do tipo árvore geralmente têm uma melhor relação *quantidade de registros / espaço físico ocupado*, pois conseguem se adaptar dinamicamente às inserções e remoções de registros em seu espaço de armazenamento.

Já as estruturas que utilizam a técnica de alocação de registros por endereçamento, como os *Grid Files,* não necessitam de reorganizações críticas de suas estruturas devido às operações executadas sobre elas, já que não usam registros como parâmetros de alocação.

Apenas em raras situações, estas estruturas precisam fazer pequenas alterações de seus espaços de endereçamento de registros através de uniões ou divisões de blocos dos espaços. Porém, estas operações não demandam altas quantidades de ciclos de máquina, pois são de natureza constante em relação ao rearranjo de registros em disco. Por outro lado, demandam um maior espaço de armazenamento para gerenciar os seus registros, devido à sua natureza semi-estática com relação aos valores dos registros que elas comportam.

3.4. *Grid Files*

Os *Grid Files* baseiam-se em representações k-dimensionais de objetos mapeados em disco, e esta estrutura origina o que nós chamamos de grades de dados. Uma grade é composta de várias células e cada uma destas células tem capacidade para armazenar um certo número de registros (c). O que ocorre é que, no funcionamento de um arquivo baseado no modelo do *Grid File,* são feitas inserções e remoções de registros que ocasionam aumentos e diminuições do tamanho da grade, para que a mesma se adapte para receber ou eliminar registros.

A operação de particionamento da grade pode ocorrer de dois modos: o primeiro e mais simples, seria no caso em que duas ou mais células da grade apontam para um mesmo *bucket* de dados no disco (*bucket* composto). Neste caso, basta se criar um novo *bucket* no disco e fazer com que uma das células que pertencem à área do *bucket* que sofreu o *overflow* aponte para este novo *bucket*, fazendo a devida transposição dos registros associados à célula escolhida para o novo *bucket*. O segundo modo de particionamento é aquele em que

a área do *bucket* que sofreu *overflow* corresponde somente a uma única célula da grade (*bucket* simples). Desta maneira a solução para o problema não é tão trivial. Toda a estrutura da grade sofrerá um particionamento em uma de suas dimensões, ocasionando inúmeras operações de entrada/saída para o reposicionamento dos registros afetados no disco.

A Figura 3.5 mostra um exemplo de *Grid File* com três dimensões (X, Y e Z) que é particionado no eixo Y, devido à sobrecarga de um *bucket* simples relativo a uma célula que se encontrava em alguma posição do espaço da grade definido pela fórmula [X_i, Y_1, Z_j], com i = [0, 1, 2] e j = [0, 1].

Figura 3.5. *Particionamento de um Grid File de 3 dimensões.*

3.5. Políticas de Divisão de *Grid Files*

A política de divisão proposta no artigo [3] é, de certa maneira, a mais simples implementada, pois a cada sobrecarga do *bucket* cuja área corresponde a um único bloco do diretório (*bucket* simples), o intervalo da grade é dividido exatamente ao meio, na respectiva dimensão escolhida. Isto torna o processo de divisão de grades extremamente rápido, mas por outro lado ineficaz, já que é um método não-otimizado e pode requerer várias

outras divisões da grade consecutivas, devido a novas inserções de registros.

De acordo com o levantamento feito durante o trabalho de elaboração da presente dissertação, em certas estruturas de arquivos, do tipo árvore, a criação de uma nova política de divisão de nós revelou grande eficiência em relação ao desempenho geral da estrutura, como em [9].

Isto aponta para a necessidade de pesquisar novas políticas de divisão para os *Grid Files*, já que este tipo de estrutura de arquivo tem uma boa escalabilidade e desempenho em muitos tipos de aplicações, principalmente em aplicações que manipulam dados geográficos, mas precisa de políticas de divisão mais inteligentes para melhorar ainda mais o seu desempenho, principalmente em situações de muitas inserções de registros.

Observa-se também em [10] que a divisão dos nós desta estrutura, descendente da KD-Tree, também tem um papel muito importante para o desempenho geral do LH*, estrutura adaptada para bancos de dados distribuídos.

A adequação das novas políticas de divisão no *Grid File* propostas neste trabalho, para aplicações que envolvem dados geográficos, como Sistemas de Informações Geográficos, podem trazer também ganhos notáveis no desempenho desta estrutura de arquivos como um todo.

Capítulo 4

Particionamento Eficiente de Grid Files em Situações de Sobrecarga de Buckets

4.1. Introdução

Este trabalho consiste em desenvolver políticas de divisão para o *Grid File* mais apropriadas para aplicações que requerem constante tratamento de objetos espaciais. Um protótipo destas políticas foi implementado para testes práticos e comparações de desempenho com outras propostas.

As propostas apresentadas neste trabalho são mais adequadas para diversas situações de uso real dos *Grid Files*. De uma maneira geral, estas situações podem ser divididas em duas classes. A primeira caracteriza-se pela necessidade de uma maior precisão no algoritmo de divisão de células, mesmo que para isso o algoritmo consuma mais ciclos de máquina.

Baseado nisso, foi desenvolvida uma política em que a grade é particionada de acordo com o ponto que garante a distribuição uniforme dos registros do bloco, ou seja, o ponto que divide os registros contidos naquela célula igualmente entre os dois novos *buckets* correspondentes às células geradas. Esta política é mostrada de uma maneira mais detalhada na Seção 4.2.1.

A segunda classe de situações pode ser caracterizada pela necessidade de um algoritmo

de particionamento que consuma a menor quantidade de processamento possível, mesmo que para isso ele não divida a grade no ponto exato de distribuição uniforme dos registros, mas num ponto próximo a este. Para isso, usamos o conceito de Centro de Massa para calcular uma boa aproximação do ponto "ótimo" para divisão da grade cujo *bucket* está sobrecarregado. Este algoritmo é descrito na Seção 4.2.2.

É preciso que se observe a importância de um bom algoritmo para encontrar o ponto de distribuição uniforme dos registros dentro de um determinado bloco em uma dada dimensão espacial. Definir tal ponto é crucial para o desempenho da política proposta e, conseqüentemente, para o desempenho da própria estrutura de arquivos como um todo.

O algoritmo proposto em [5] para executar o particionamento da grade é bem simples, pois apenas divide a grade no ponto médio da célula sobrecarregada em uma determinada dimensão. Contudo, é pouco eficiente, pois, em determinadas situações (onde há agrupamentos de registros em cantos da célula, por exemplo), ocasionará uma freqüência de divisões muito alta, pois uma das duas novas células ficará com uma quantidade ainda muito alta de registros (próxima ao limite máximo de registros), e se houver mais inserções nesta célula, ela provavelmente deverá ser novamente particionada.

4.2. Algoritmos Propostos

4.2.1. Algoritmo da Média dos Elementos Medianos

O algoritmo da Média dos Elementos Medianos (MEM) deve particionar uma célula da grade cujo *bucket* por ela referenciado está sobrecarregado. Em outras palavras, o *bucket* que contém os registros desta célula encontra-se com a sua capacidade máxima de registros (c) alcançada. Portanto, são conhecidos exatamente quantos pontos (registros) existem na célula e esses registros encontram-se em memória.

Como já visto anteriormente, existem dois casos a serem analisados no processo de particionamento de *bucket*. Isto porque a área do *bucket* pode corresponder a várias células da grade ou a apenas uma célula. No primeiro caso, basta que se aloque um novo *bucket*

no disco, que será referenciado por uma das células da área sobrecarregada e dividam-se os registros do *bucket* sobrecarregado com este novo *bucket*, de acordo com a posição dos registros dentro das células.

Para o segundo caso, quando a área do *bucket* sobrecarregado é formada por uma única célula, deve-se particionar a célula sobrecarregada, ou seja, deve particionar-se a grade em alguma dimensão, de maneira que o ponto de particionamento se situe dentro do intervalo da célula sobrecarregada, isto é, da célula cujo bucket encontra-se sobrecarregado. Para calcular este ponto, propõe-se o algoritmo MEM.

O algoritmo MEM funciona como descrito a seguir:

Passo 1. Ordenar os pontos da célula utilizando-se o algoritmo Quick-Sort. O critério de ordenação é definido pela dimensão escolhida para o particionamento. Por exemplo, se a dimensão escolhida é a do eixo x, os pontos devem ser ordenados pelo valor da coordenada x.

Passo 2. Em relação à dimensão que se particionará, recuperar os valores das coordenadas dos pontos situados como $\left\lceil \frac{n}{2} \right\rceil$ e $\left(\left\lceil \frac{n}{2} \right\rceil + 1\right)$-ésimos elementos da seqüência ordenada, onde n = c (capacidade máxima de registros de cada *bucket*).

Passo 3. Calcular a média aritmética simples destas duas coordenadas. Será obtido como resultado deste cálculo, o valor da coordenada onde será feito o particionamento da grade.

Passo 4. Particionar a grade na coordenada calculada no passo anterior de acordo com a dimensão escolhida para o particionamento.

Passo 5. Realocar os registros existentes no *bucket* sobrecarregado nos *buckets* correspondentes a cada uma das duas novas células geradas.

É importante notar que outras células são envolvidas no processo de particionamento, apesar de não estarem sobrecarregadas. Isto acontece porque o particionamento é realizado em toda a extensão da grade, e não só na célula sobrecarregada. Contudo, estas

células que não encontravam-se com seus *buckets* sobrecarregados, continuam referenciando o mesmo *bucket*. Não há necessidade de alocação de novos *buckets* em disco para tais células, já que não houve sobrecarga nos *buckets* referenciados por estas células.

Figura 4.1. *Política de divisão proposta para o Grid File segundo o algoritmo da Média dos Elementos Medianos*

A Figura 4.1 ilustra o funcionamento da política proposta em um *Grid File* de 2 dimensões cuja área do *bucket* sobrecarregado é constituída por uma única célula. Observe que os pontos estão agrupados em uma determinada região da célula a ser dividida[1]. Esta situação representa o pior caso para o algoritmo proposto em [5]. Considere uma capacidade "c" de *bucket* igual a 8 (oito). As coordenadas dos pontos são P_1(4,13), P_2(5,18), P_3(6,12), P_4(7,16), P_5(9,14), P_6(10,19), P_7(12,11) e P_8(13,17) e a célula está delimitada inicialmente por $x1 = 0$, $x3 = 30$, $y1 = 0$ e $y3 = 20$. Segundo a estratégia proposta, estes pontos serão ordenados pelo método Quick-Sort de acordo com a dimensão mais apropriada (Passo 1). Caso a dimensão escolhida para o particionamento seja a do eixo X, então a seqüência de pontos ordenados será a seguinte: P_1, P_2, P_3, P_4, P_5, P_6, P_7 e P_8. Caso a dimensão escolhida seja a do eixo Y, então a ordenação será a seguinte: P_7, P_3, P_1, P_5, P_4, P_8, P_2 e P_6.

[1]Os pontos P_1, P_2, P_3, P_4, P_5, P_6, P_7 e P_8 estão próximos do canto superior esquerdo da célula.

De acordo com o algoritmo descrito, o próximo passo é determinar os $\left\lceil \frac{n}{2} \right\rceil$ e $\left(\left\lceil \frac{n}{2} \right\rceil + 1\right)$-ésimos elementos da seqüência ordenada. Sabemos que n = c = 8, portanto $\left\lceil \frac{n}{2} \right\rceil = \left\lceil \frac{8}{2} \right\rceil = 4$ e $\left(\left\lceil \frac{n}{2} \right\rceil + 1\right) = \left(\left\lceil \frac{8}{2} \right\rceil + 1\right) = (4+1) = 5$. É importante ressaltar que, independentemente da dimensão escolhida para o particionamento, os pontos a serem escolhidos sempre serão o 4° e o 5° da seqüência ordenada.

Suponha que o particionamento seja em relação ao eixo X. Nesse caso, o quarto elemento é o ponto P_4 e o quinto elemento é o ponto P_5. Os valores das coordenadas x dos pontos P_4 e P_5 são 7 e 9, respectivamente. Efetuando-se a média aritmética (Passo 3), teremos então: $\frac{7+9}{2} = \frac{16}{2} = 8$, que será tomado como ponto de divisão da célula (Passo 4) no eixo X (ponto x2 na Figura 4.1). Os registros são então redistribuídos pelas duas células resultantes da divisão da célula inicial (Passo 5).

Agora suponha um particionamento no eixo Y. Neste caso, os elementos selecionados seriam os pontos P_5 e P_4, cujas coordenadas, no eixo Y, são 14 e 16, respectivamente. Logo, feita a média aritmética, obteremos o valor 15, coordenada do eixo Y que deverá ser usada para definir o hiperplano de particionamento (ponto y2).

O caso em que a área do *bucket* sobrecarregado corresponde a mais de uma célula já foi citado na Seção 3.4 da revisão bibliográfica.

Para que se aprecie a importância de uma boa política de divisão de grades no *Grid File*, podemos citar "O splitting não é, portanto, uma operação local e pode conduzir ao crescimento exponencial do diretório da grade mesmo para distribuição uniforme dos dados", retirado de [11].

4.2.2. Algoritmo do Centro de Massa

Uma outra alternativa proposta nesse trabalho para determinar o ponto que será usado para particionar a célula que se encontra sobrecarregada é utilizar o conceito de centro de massa.

Considere um conjunto de pontos P_1, P_2, P_3, ..., P_n com massas m_1, m_2, m_3, ..., m_n , respectivamente. Cada um destes pontos pode ser representado, numa certa dimensão w,

como sendo $P_1(x_1, y_1, z_1, ..., w_1)$, $P_2(x_2, y_2, z_2, ..., w_2)$, $P_3(x_3, y_3, z_3, ..., w_3)$, ..., $P_n(x_n, y_n, z_n, ..., w_n)$. O centro de massa será o ponto C de coordenadas (x_{CM}, y_{CM}, z_{CM}, ..., w_{CM}), obtidas através das médias ponderadas:

$x_{CM} = \frac{m_1x_1+m_2x_2+...+m_nx_n}{m_1+m_2+...+m_n}$

$y_{CM} = \frac{m_1y_1+m_2y_2+...+m_ny_n}{m_1+m_2+...+m_n}$

$z_{CM} = \frac{m_1z_1+m_2z_2+...+m_nz_n}{m_1+m_2+...+m_n}$

...

$w_{CM} = \frac{m_1w_1+m_2w_2+...+m_nw_n}{m_1+m_2+...+m_n}$

O algoritmo baseia-se nos pontos pertencentes à célula em questão. No caso geral, atribui-se massa igual a 1 (um) a cada um dos pontos ($m_i = 1$; $i = 1...n$) e calcula-se o seu centro de massa, de acordo com a dimensão escolhida para o particionamento. O centro de massa para a dimensão X, por exemplo, é calculado usando-se a seguinte fórmula:

CM = $\frac{m_1x_1+m_2x_2+...+m_nx_n}{m_1+m_2+...+m_n}$; como sabemos, podemos substituir o valor de qualquer m por 1, então teremos:

CM = $\frac{1x_1+1x_2+...+1x_n}{1+1+...+1}$, logo teremos a seguinte fórmula:

CM = $\frac{x_1+x_2+...+x_n}{n}$, que pode ser calculada para qualquer outra dimensão da grade, substituindo apenas a variável x pela variável correspondente a dimensão escolhida.

O caso descrito anteriormente e ilustrado na Figura 4.1, pode ser usado como exemplo do funcionamento desta política. No caso do particionamento ser feito sobre o eixo X, então o centro de massa deverá ser calculado segundo a fórmula CM = $\frac{x_1+x_2+...+x_n}{n}$. Usando-se os valores definidos no exemplo, tem-se então CM = $\frac{4+5+6+7+9+10+12+13}{8} = \frac{66}{8} = 8,25$, que será o valor da coordenada de particionamento no eixo X. Caso a dimensão mais apropriada para o particionamento seja a do eixo Y, então tem-se a seguinte expressão: CM = $\frac{y_1+y_2+...+y_n}{n}$, e substituindo-se os valores de y: obtem-se CM = $\frac{13+18+12+16+14+19+11+17}{8} = \frac{120}{8} = 15$, ponto do eixo Y no qual se fará o particionamento da grade.

Pode-se observar que, em alguns casos, esta aproximação é uma boa solução para o

problema de particionamento. Além disso, sua ordem de complexidade pode ser considerada inferior a do primeiro algoritmo, pois, neste último caso, não é necessário fazer a ordenação dos registros. Basta conhecer os valores de suas coordenadas para se calcular o centro de massa.

Contudo, em alguns casos, este algoritmo não resolve o problema do particionamento adequadamente, como podemos observar pelo exemplo da Figura 4.2.

Figura 4.2. *Política de divisão proposta segundo o algoritmo do Centro de Massa*

Neste caso, temos um *Grid File* com capacidade de *bucket* "c" igual a 6, com os pontos P_1(4,16), P_2(9,16), P_3(7,15), P_4(5,14), P_5(10,13) e P_6(28,1), além da célula estar delimitada inicialmente por $x1 = 0$, $x3 = 30$, $y1 = 0$ e $y3 = 20$. Aplicando-se o algoritmo do Centro de Massa para o particionamento na dimensão do eixo X, obteremos CM = $\frac{4+9+7+5+10+28}{6} = \frac{63}{6} = 10,50$. Como podemos observar, esta coordenada do eixo X não é a mais apropriada para dividir os pontos contidos nesta célula uniformemente. No caso do particionamento ser feito no eixo Y, o algoritmo também não obterá a melhor coordenada para a divisão, pois, seguindo os cálculos, obteremos CM = $\frac{16+16+15+14+13+1}{6} = \frac{75}{6} = 12,50$, coordenada esta que não particiona o eixo Y da maneira mais uniforme em relação aos pontos (registros) contidos naquela célula.

Devemos observar, portanto, que, ao contrário da primeira alternativa, esta política

não se mostra eficaz para todas as situações possíveis de divisão de células em um *Grid File*.

Capítulo 5

Descrição do Protótipo para Gerenciar Dados Armazenados em Grid Files

5.1. Introdução

A eficiência de uma estrutura de dados é determinada por dois fatores essenciais: o tempo de processamento para acessar dados na estrutura e a utilização do espaço de memória. Uma estrutura de dados eficiente deve procurar um ponto de equilíbrio entre estes dois fatores. Dessa forma, a estrutura será adequada para vários tipos de aplicações, com as mais variadas necessidades de tempo de resposta e diante de uma diversidade de tamanhos de conjuntos de dados (geralmente de tamanho considerável em SIG's, podendo ir de alguns gigabytes a até terabytes de informações).

A estrutura *Grid File* foi projetada para minimizar o número de acessos ao disco, reduzindo, dessa forma, o tempo de processamento para acessar dados armazenados em *Grid Files*. Adicionalmente, a estrutura de *Grid Files* garante uma boa utilização do espaço de armazenamento disponível. Isto é possível graças ao conceito de *buckets* de disco. Um *bucket* pode ser totalmente recuperado (carregado) em uma página de memória principal. Políticas eficientes de união e de divisão de células têm sido propostas para dar suporte a um gerenciamento eficiente de alocação de espaço de armazenamento dos dados.

Uma política eficiente de divisão deve garantir um bom resultado no particionamento de células da grade, levando-se em consideração dois aspectos. Em primeiro lugar, precisa ser rápida o suficiente para não impactar negativamente no desempenho geral da estrutura. Em segundo lugar, deve ser eficiente ao ponto de distribuir uniformemente os registros entre os *buckets* após a divisão, objetivando minimizar o número total de divisões que a grade sofrerá para realizar uma determinada tarefa, como, por exemplo, inserir vários registros em uma certa região.

Para comprovar a eficiência das políticas de particionamento propostas nessa dissertação (veja Seções 4.2.1 e 4.2.2), foi desenvolvido um sistema capaz de simular o funcionamento da estrutura de arquivos de *Grid Files* usando essas políticas. Este capítulo está organizado como descrito a seguir. Na Seção 5.2, são descritas as principais características do protótipo implementado. Em seguida, na Seção 5.3, é mostrado como estão implementadas as rotinas do sistema de simulação de *Grid Files*. Além disso, explica-se como funciona o protótipo implementado em Java e é feita uma descrição passo a passo de como o mesmo deve ser usado da perspectiva de um usuário comum.

Na Seção 5.4, são descritos os parâmetros de qualidade a serem alcançados pelos algoritmos de divisão de células propostos, para que eles sejam considerados como melhores alternativas ao uso do algoritmo de divisão tradicional. E, finalmente, na Seção 5.5 são mostrados os resultados alcançados pelos algoritmos propostos nessa dissertação, com base no modelo de *Growing File*, onde a estrutura do arquivo sofre constantes inserções de registros e, assim, torna-se mais evidente o algoritmo que tem maior capacidade de absorver uma grande quantidade de registros, sem gerar um elevado número de particionamentos da estrutura de *Grid Files*. Os resultados são mostrados como tabelas e gráficos, para uma maior clareza dos resultados obtidos pela comparação dos algoritmos propostos com o algoritmo tradicional.

5.2. Estratégias de Projeto

O protótipo desenvolvido simula uma estrutura de *Grid File* na qual o processo de divisão de *buckets* deve ser feito através da utilização de uma das políticas propostas apresentadas no Capítulo 4 (Seções 4.2.1 e 4.2.2) ou da política de particionamento tradicional de *Grid Files* (Seção 3.5).

O objetivo da simulação apresentada é avaliar o desempenho das duas estratégias propostas neste trabalho para dividir as células do *Grid File*, compará-las com outras propostas de particionamento, e verificar a corretude do modelo proposto.

Durante a fase de implementação, foi necessário, inicialmente, definir a estratégia a ser utilizada pelo sistema no caso da sobrecarga de uma célula da grade. Existem duas alternativas a serem tomadas neste aspecto, tais como: se as duas estratégias propostas deveriam ser implementadas conjuntamente em um mesmo sistema ou se ficariam em sistemas separados, totalmente independentes uma da outra.

Escolhida a abordagem de implementar as duas políticas em um só programa, decidiu-se também que o usuário do sistema teria a capacidade de escolher qual a melhor política de *splitting* para cada caso ou se ele apenas indicaria uma política a ser adotada e esta seria usada durante todo aquele ciclo de execução do sistema. Uma outra abordagem seria configurar o sistema para que ele fosse "inteligente" o suficiente para decidir qual a melhor estratégia a ser usada em cada caso, mas esta abordagem pressupõe metodologias de implementação que estão além do escopo desta dissertação.

Foram implementadas as duas estratégias de *splitting* propostas neste trabalho, além da estratégia de *splitting* tradicional, em um mesmo sistema, permitindo ao usuário selecionar individualmente a política desejada em cada caso de *bucket overflow*. Desta maneira, o sistema foi implementado mais rapidamente, e com a flexibilidade necessária para testar vários padrões de uso para o mesmo, podendo alternar entre as políticas propostas ou usar uma única política durante todo um ciclo do programa, verificando e comparando a velocidade de execução e o desempenho geral do sistema em cada uma das alternativas escolhidas.

5.3. Descrição do Protótipo

O protótipo para teste das políticas de *Grid Files* foi escrito em Java e consta de, aproximadamente, 1160 linhas de código.

A implementação, inicialmente, havia sido feita em linguagem C, devido ao seu excelente desempenho em termos de velocidade de execução, além de que o SGBD Informix® também tem suporte para receber módulos DataBlades nesta linguagem. Porém, encontramos diversas dificuldades com o tratamento do gerenciamento de memória, já que a linguagem C não apresenta um bom suporte a este aspecto da programação e execução de programas, deixando totalmente a cargo do programador da aplicação a responsabilidade de fazer este trabalho de gerenciamento de alocação e liberação do espaço de memória disponível na máquina em que o sistema está sendo executado, o que dificultaria bastante o trabalho de implementação.

A partir deste fato, decidimos converter toda a implementação escrita em C para a linguagem Java, já que esta apresenta uma característica conhecida como *Garbage Collection*, onde o gerenciamento do espaço de memória (liberação) é feito automaticamente pelo sistema, ficando o programador livre desta tarefa. Além disso, o uso da linguagem Java faz com que o sistema fique independente de plataforma, pois esta é uma das principais características desta linguagem, podendo o mesmo programa ser executado tanto em um micro PC como em uma estação UNIX. Vale ressaltar ainda que foi crucial para esta decisão ter conhecimento de que programas em Java também são aceitos como módulos DataBlades do Informix.

Para a execução dos testes, foi usada uma máquina PC com processador K6-II 500 MHz, com 128 MB de memória principal, tendo como sistema operacional o sistema Microsoft Windows 2000® Professional (versão 5.0.2195 SP2). A interface do sistema em modo texto contribuiu para a melhoria do desempenho da máquina executando o protótipo implementado.

Esta implementação poderá servir de base para a criação de um módulo DataBlade do Banco de Dados Informix®. A idéia foi a de estender o Informix para suportar um

Sistema de Informação Geográfica.

O programa, com interface do tipo texto, funciona da seguinte maneira:

Na tela inicial, o usuário é informado sobre as configurações pré-definidas do programa. São elas: número de dimensões do *Grid File*, capacidade máxima dos *buckets* e quais os valores (limites) das escalas lineares definidas para cada uma das dimensões. Os testes foram feitos utilizando-se um *Grid File* com 3 dimensões (X, Y e Z) e capacidade de *bucket* igual a 3 registros. As escalas lineares foram definidas inicialmente em X(0, 500, 1000, 1500, 1750, 1875, 2000), Y(1, 4, 6, 10, 14, 17, 20, 26) e Z(1, 2, 3, 4, 5, 6, 7).

Para se inserir registros no *Grid File,* existem duas possibilidades. A primeira é fazer a inserção manual, registro a registro. Para isso, deve-se informar quais os valores dos atributos usados como chave de indexação (escalas ou dimensões da grade) do registro que se quer inserir. A segunda opção é realizar a inserção de maneira automática, através da criação randômica de registros. Para isso, o sistema solicita ao usuário que informe se deseja usar a entrada automática (randômica) de dados.

Em seguida, se o modo de entrada automática for selecionado, é requisitado o número de registros randômicos que se deseja que sejam gerados e armazenados no *Grid File*. O próximo passo é determinar, de forma interativa, o algoritmo de particionamento em caso de sobrecarga de *bucket*. As opções são as seguintes:

1. Solicita escolha do usuário a cada sobrecarga
2. Média dos Elementos Centrais
3. Centro de Massa
4. Divisão Tradicional

Em seguida, o sistema faz a inserção dos registros criados randomicamente na estrutura de armazenamento *Grid File*.

O sistema faz a busca pelo *bucket* correspondente à célula que contém o registro a ser inserido e cria, automaticamente, os *buckets* para as células que ainda não possuem seus respectivos *buckets* em disco.

Se o sistema detectar que se está tentando inserir um registro em um *bucket* que se encontra com sua capacidade máxima atingida, é disparado o procedimento de *splitting*. Por sua vez, este procedimento pode disparar o processo de particionamento de células, caso o *bucket* sobrecarregado seja apontado por uma única célula, isto é, corresponda a um *bucket* simples (Seção 1.1).

Se houver particionamento de células e o algoritmo de particionamento tiver sido pré-selecionado, o sistema fará o particionamento da grade de acordo com este algoritmo. No caso em que o usuário tenha selecionado fazer a escolha pelo algoritmo de divisão a ser utilizado em cada sobrecarga, o sistema exibe um menu de opções para que o usuário escolha qual o melhor método para particionar a célula sobrecarregada em questão, sendo as opções deste menu as seguintes:

1. Média dos Elementos Centrais
2. Centro de Massa
3. Divisão Tradicional

Os procedimentos e métodos do programa foram implementados de maneira a não deixar dúvidas sobre sua utilização, fazendo, para isso, entradas e saídas com tipagens bem caracterizadas através do uso da nomenclatura das estruturas e objetos que compõem o sistema. A Tabela 5.1 mostra o esquema (assinatura) dos principais métodos da implementação feita.

Para fins ilustrativos, um pseudo-código do procedimento que particiona a estrutura de *Grid Files* é mostrado a seguir:

```
proc particionar_grid(ponto, bucket, dimensao)
/* É passado qual o ponto a ser feito o particionamento*/
{
   /* Neste procedimento deve-se dividir todo o Grid File de
   acordo com a dimensão e a posição passadas como parâmetros,
```

Método	Entrada	Saída
localizar_bucket	float x, float y, float z	Bucket
novo_registro	float x, float y, float z	Registro
inserir_registro	Bucket b, Registro r, int opDefault	Bucket
coordenada2posicao	float x, float y, float z	void
criar_bucket	int coordX, int coordY, int coordZ	Bucket
dividir_bucket	Bucket buc, int opcao	Bucket
particionar_grid	float ponto, Bucket bucket, int opcao	void
realocar_bucket	Bucket b1, Bucket b2, float ponto, int opcao	void
recuperar_registro	Bucket b, int rid	Registro
quicksort	float a[], int lo0, int hi0	void

Tabela 5.1. *Assinatura dos métodos do protótipo de Grid Files.*

```
   realocando-se os registros */
   caso (dimensao)
   {
      seja X: /* Ajustar o particionamento do espaço da grid
      na escala linear correspondente (Eixo X) */
      {
//       INÍCIO do particionamento e ajuste da escala linear
          Incrementar o vetor X de uma posição;
          Deslocamento à direita dos elementos do vetor X
             deste o final até a posição de particionamento;
          Incluir o ponto de particionamento no vetor X;
          Incrementar o contador do tamanho da escala
             linear (vetor X) da dimensão X (sX);
//       FIM do particionamento e ajuste da escala linear

//       INÍCIO do particionamento e ajuste do grid directory
          Deslocamento à direita em relação ao eixo X dos
```

```
                elementos da matriz desde a última posição até
                o ponto de particionamento;
//          As células no limite do particionamento devem
//            apontar para o mesmo bucket que suas células
//            originais.
            Criar (alocar) um novo bucket para a célula que
                teve o respectivo bucket sobrecarregado;
            Inserir o registro desejado na célula com o novo
                bucket alocado;
            Rearranjar os registros da célula sobrecarregada
                entre as duas novas células geradas.
            Sair do case;
        }
        seja Y:
        {
            /* Repetir o procedimento do caso anterior
        para ajustar o particionamento do espaço da grid
        na escala linear correspondente (Eixo Y) */
        }
        seja Z:
        {
            /* Repetir o procedimento do caso anterior
        para ajustar o particionamento do espaço da grid
        na escala linear correspondente (Eixo Z) */
        }
        default: Imprimir("** Erro: Nao existe a
                  dimensão requerida para a operação
                  de particionamento! ** \n");
```

```
                Sair;
    }
} // particionar_grid()
```

O código-fonte (em Java) referente ao pseudo-código mostrado acima encontra-se no Apêndice A desta dissertação e corresponde ao método *particionar_grid()* (linhas 407 à 803).

O algoritmo de particionamento mostrado acima funciona, em linhas gerais, da seguinte maneira:

1. Seleciona-se qual a dimensão em que será feito o particionamento a partir da variável *dim* (dimension);

2. Insere-se o novo ponto de particionamento (*ponto*) no array que representa a escala linear da dimensão selecionada através de um *shift right* no array a partir da última posição do array até a posição onde o novo limite (ponto de particionamento) será inserido;

3. As alterações realizadas na escala linear são refletidas na estrutura (matriz de 3 dimensões) de células do *Grid File* (*grid[][][]*), através do deslocamento das células e da alocação de novo(s) bucket(s) para armazenar os registros da célula sobrecarregada;

4. Realiza-se, finalmente, a realocação dos registros no novo bucket gerado na estrutura matricial de células do *Grid File*, através da chamada ao método *realocar_bucket()*.

O algoritmo usado para ordenar os registros dentro do *bucket*, em caso de divisão, foi o QuickSort, por ser um algoritmo eficiente ($O(n \cdot \log_2 n)$). Caso fosse adotado um algoritmo ineficiente para realizar a ordenação, o desempenho da política de divisão proposta na Seção 4.2.1 poderia ser comprometido.

É mostrado, a seguir, o procedimento criado para inserir um registro r em um *bucket* b, testando se a capacidade máxima da célula em questão foi ou não atingida.

```
proc inserir_registro(bucket, registro, opcao)
{
   Se a capacidade máxima do bucket ainda não tiver
      sido atingida, então
   {
      Adicionar registro ao bucket;
      Incrementar o contador de registros do bucket;
   }
   Senão, se o bucket sobrecarregado for um bucket
      composto, então:
   {
      Alocar um novo bucket;
      Dividir as células da região do bucket sobrecar-
         regado com o novo bucket alocado;
      Rearranjar os registros do bucket sobrecarregado
         entre as novas regiões dos dois buckets;
   }
   Senão
   {
      Chamar o método dividir_bucket();
      Inserir registro no novo bucket;
   }
} // inserir_registro()
```

O código em Java referente ao pseudo-código mostrado acima encontra-se no Apêndice A e corresponde ao método *inserir_registro()* (linhas 1006 à 1072).

Feita a escolha do usuário, o sistema dispara os procedimentos necessários para particionar a grade no ponto determinado de acordo com a estratégia escolhida.

O procedimento que calcula o ponto de divisão mais apropriado, para uma certa

dimensão, é exibido através do pseudo-código a seguir.

```
proc dividir_bucket(bucket, opcao)
{
   Escolher a dimensão a ser particionada de acordo com o
      método FIFO;
   Se o algoritmo escolhido for o MEM - Média dos
      Elementos Centrais - então
   {
      Usar o metodo Quick-Sort para ordenar os elementos
         (registros) dentro do bucket específico;
      Calcular os índices dos dois registros medianos do
         bucket em relação a coordenada escolhida para o
         particionamento;
      Calcular o ponto médio como sendo a média aritmética
         das coordenadas dos registros contidos nos índices
         calculados no passo anterior;
   }
   Senão, se o algoritmo escolhido for o do Centro de
      Massa, então
   {
       Calcula-se o ponto médio como sendo o centro de
          massa das coordenadas dos registros contidos
          no bucket sobrecarregado;
   }
   Senão, se o algoritmo escolhido for o de divisão
      tradicional, então
   {
      Calcula-se o ponto médio como sendo o meio do
```

```
            intervalo da célula que referencia o bucket
            sobrecarregado;
      }
      Incrementat o contador de divisões;
      Chamar o método particionar_grid(), passando o
         ponto médio calculado;
} // dividir_bucket()
```

O código Java correspondente ao código mostrado acima é encontrado no Apêndice A, no método de mesmo nome (linhas 817 à 923).

O protótipo implementado apresenta ainda um módulo de consulta. Nesse módulo, deve-se informar as coordenadas de um ponto, a respeito do qual se deseja visualizar informações. Caso existam dados para o ponto consultado, o sistema localizará o *bucket* que contém estes dados, exibindo o código identificador do *bucket* e todos os pontos contidos no mesmo. Caso o ponto não exista, o sistema emitirá uma mensagem informando ao usuário que o ponto desejado não existe.

Analisando o desempenho da aplicação, observou-se que a estratégia de divisão através do cálculo do Centro de Massa (CM) tem uma pequena vantagem sobre o método da Média dos Elementos Medianos (MEM) em relação à velocidade de processamento, já que este último método necessita que seja feita a ordenação dos registros dentro do *bucket*, e para isso utilizou-se o método de ordenação Quick Sort, que tem complexidade da ordem de (n . $\log_2$ n), a qual é maior de que a complexidade do primeiro algoritmo (CM) que é linear (O(n)).

Portanto pode-se observar que o algoritmo MEM devolve sempre o ponto de divisão mais uniforme para os registros da célula sobrecarregada, mas ao mesmo tempo consome um pouco mais dos recursos de máquina para isso, enquanto que o algoritmo CM retorna um ponto de divisão aproximado do ponto de divisão mais uniforme.

5.4. Parâmetros e Modelos de Testes do Protótipo Implementado

Os testes práticos do protótipo de sistema de arquivos multi-indexados (*Grid Files*) implementado envolvem um estudo comparativo entre as estruturas de arquivos multi-indexados já implementadas e a implementação do *Grid File* feita nesse trabalho. Para executar estes testes, primeiramente precisam-se definir as seguintes características básicas a serem alcançadas pelo sistema:

- Melhor tempo de execução do algoritmo de divisão de células;
- Melhor eficiência do algoritmo de divisão, ou seja, menor número de divisões para executar uma determinada quantidade de inserções de registros;
- Maior escalabilidade da estrutura de arquivo.

Para a execução dos testes em sistemas de arquivos para medir o desempenho de nossa aplicação, foram levados em conta três modelos diferentes que caracterizam, de uma maneira geral, as mais variadas situações de uso real do *Grid File* que podem ocorrer para este tipo de sistema:

- *growing file* (ou arquivo em crescimento): Neste modelo de funcionamento do sistema, o arquivo recebe várias inserções de registros repetidamente, crescendo rapidamente;
- *steady-state file* (ou arquivo estático): Neste caso, durante a execução do sistema, há tantas inserções quanto remoções, assim o número de registros no arquivo é mantido aproximadamente constante;
- *shrinking file* (ou arquivo em contração): Esta classe de simulação é aquela onde os arquivos sofrem um número muito maior de remoções de registros em sua execução do que de inserções, fazendo o tamanho do arquivo diminuir vertiginosamente.

Para o estudo de caso das políticas de divisão propostas, os modelos mais apropriados são o *growing file* e o *steady-state file*, pois nestes tipos de aplicações acontece uma maior requisição de particionamentos de células, ficando o *shrinking file* mais indicado para testes em políticas de união de células da grade. Apesar disso, executamos nossos testes utilizando o modelo de *growing file*, pois este é o modelo que oferece o maior grau de dificuldade para a estrutura de arquivo testada no que diz respeito ao suporte a inserções em massa de registros.

5.5. Testes com Base no Modelo *Growing File*

Os testes foram executados utilizando o modelo *Growing File.* Isto deve-se ao fato de que este modelo é o que melhor representa uma situação crítica de uso para execução do sistema de armazenamento através de *Grid Files*, quando pretendemos mensurar e comparar as políticas de divisão implementadas.

Os testes sobre o protótipo da estrutura de *Grid Files* implementado em Java confirmaram as expectativas com relação a eficiência das políticas de divisão propostas e ainda revelaram alguns detalhes interessantes a respeito do comportamento dos *Grid Files* frente ao modelo de arquivos em crescimento (*Growing Files*). Os testes foram realizados usando a entrada automática (randômica) de registros, para simular o funcionamento real de um SGBD com muitas inserções de registros simultaneamente. É importante ressaltar também que o espaço de endereçamento de *buckets* máximo permitido, com a grade 3-dimensional, foi de 39 células em cada dimensão, pois o *grid directory* ficou definido como uma matriz de 40 por 40 por 40 elementos.

Os testes foram divididos em duas categorias para melhor observar o desempenho da estrutura: a primeira levou em consideração o comportamento diante do modelo *Growing File* com até 10.000 inserções de registros. Nesta situação, as três políticas comparadas - as duas propostas neste trabalho e a política tradicional de divisão - tiveram resultados semelhantes, tanto em relação ao desempenho, medido em função do tempo de processamento das inserções dos registros, quanto em relação ao número de divisões executadas

para inserir uma determinada quantidade de registros criados randomicamente na estrutura.

Tabela 5.2. *Tempo de execução do algoritmo por número de registros inseridos.*

Tempo\Reg.	200	400	600	800	1000	2000	4000	6000	8000
MEM	230	230	230	251	250	290	361	470	571
CM	210	210	221	230	241	260	350	421	501
Tradicional	201	211	220	241	230	270	371	441	561

A Tabela 5.2 acima mostra o tempo de execução (em milisegundos) de cada algoritmo comparado através da mensuração feita por diferentes quantidades de registros inseridos nos testes, variando os pontos de teste entre 200 a até 8000 registros inseridos. Por exemplo, para inserir 200 registros no *Grid File*, o algoritmo MEM levou 230 milisegundos, o algoritmo CM gastou 210 milisegundos, enquanto que o algoritmo de particionamento tradicional precisou de 201 milisegundos.

Através destes testes, podemos concluir que, na média, para esta faixa de 200 a até 8000 registros inseridos, o algoritmo tradicional de divisão foi 3,86% pior de que o algoritmo CM, ao passo que foi 4,75% melhor que o algoritmo MEM em relação ao tempo de execução dos algoritmos para inserir uma mesma quantidade de registros de acordo com a tabela mostrada.

A Figura 5.1 esboça um gráfico representando as diferenças em termos de tempo de execução dos algoritmos envolvidos no teste com base nos dados da Tabela 5.2.

Tabela 5.3. *Quantidade de divisões realizadas pelo algoritmo por número de registros inseridos.*

Divisões\Reg.	200	400	600	800	1000	2000	4000	6000	8000
MEM	3	11	13	23	18	31	34	55	66
CM	3	12	13	19	24	26	39	44	52
Tradicional	7	13	15	23	20	33	43	50	70

Na Tabela 5.3, é mostrado o número de divisões que cada algoritmo realiza,

Figura 5.1. *Tempo de execução vs. Quantidade de Registros (< 10000)*

comparando-os através da mensuração feita por diferentes quantidades de registros inseridos nos testes, variando os pontos de teste entre 200 a 8.000 registros inseridos. Vale ressaltar que estes testes foram realizados conjuntamente com os testes de tempo de execução mostrados na Tabela 5.2, mas por motivos de melhor compreensão e visualização são mostrados separadamente. Pelos valores mostrados na tabela, podemos concluir que o algoritmo tradicional de divisão teve um desempenho, em média, aproximadamente 8% pior de que o algoritmo MEM e foi, também em média, aproximadamente 18% pior de que o algoritmo CM.

Abaixo é mostrado o gráfico (Figura 5.2) comparando o número de divisões de cada algoritmo com diferentes quantidades de registros inseridos (até o máximo de 8.000 inserções de registros), de acordo com a Tabela 5.3.

Vale lembrar que o objetivo das políticas propostas é diminuir o valor dos parâmetros: tempo de execução e quantidade de divisões.

Figura 5.2. *Número de Divisões vs. Quantidade de Registros (< 10000)*

Na segunda categoria de testes, o desempenho do *Grid File* foi medido usando o modelo *Growing File* variando a quantidade de registros inseridos entre 10.000 e 100.000 registros. Nessa categoria, ficou clara a superioridade das políticas de particionamento propostas sobre a política tradicional, que muitas vezes (falha de 69,44%, ou seja, executou com sucesso somente em 30,56% das vezes) não conseguiu gerenciar a inserção da grande quantidade de registros dentro do espaço de endereçamento de 39 células em cada dimensão (escala linear), enquanto que as políticas propostas executaram as inserções com grande percentual de sucesso (execução correta em 83,33% das tentativas).

Na Tabela 5.4, são mostrados alguns resultados obtidos nos testes de execução do *Grid File* com relação ao tempo de execução do algoritmo durante a inserção de mais de 10.000 registros randomicamente. Estes resultados foram encontrados depois de sucessivos testes realizados, em baterias de 3 execuções para cada quantidade de registros, onde, ao final, o valor considerado foi a média aritmética simples dos valores mensurados nas execuções.

Nota-se que, nas vezes em que o algoritmo tradicional conseguiu finalizar a sua execução, ele foi, em média, 33,9% melhor de que o algoritmo MEM e 21,6% melhor de que o algoritmo CM. Isto não significa que o algoritmo tradicional seja a melhor escolha para esta quantidade de inserções, pois ele falhou em 69,44% das vezes que foi executado, contra apenas 16,67% dos algoritmos MEM e CM.

Tabela 5.4. *Tempo de execução do algoritmo por número de registros inseridos*

Tempo\Reg.	10000	20000	40000	45000	50000	60000	80000	100000
MEM	564	1032	1943	2103	2239	2955	3576	4326
CM	581	998	1842	2076	2169	2617	3665	—
Tradicional	628	1151	—	—	—	2924	—	—

Na Figura 5.3, são ilustrados graficamente os valores contidos na Tabela 5.4.

Figura 5.3. *Tempo de execução vs. Quantidade de Registros (> 10000)*

Na Tabela 5.5, é mostrada a quantidade de divisões de cada algoritmo para inserir entre 10 mil e 100 mil registros no *Grid File* usando diversas quantidades de registros

como entrada. Observa-se que, com relação ao número de divisões efetuadas, o algoritmo tradicional foi, em média, aproximadamente 7% pior de que o algoritmo MEM e 15% pior de que o algoritmo CM.

Tabela 5.5. *Tempo de execução do algoritmo por número de registros inseridos*

Divisões\Reg.	10000	20000	40000	45000	50000	60000	80000	100000
MEM	51	70	79	83	85	96	84	87
CM	58	70	81	79	75	81	88	—
Tradicional	65	88	—	—	—	95	—	—

Na Figura 5.4, é mostrado o gráfico ilustrando as mensurações feitas na Tabela 5.5.

Figura 5.4. *Tempo de execução vs. Quantidade de Registros (> 10000)*

Com base nos testes realizados, pode-se observar que o desempenho do algoritmo do Centro de Massa (CM) é melhor do que o algoritmo de Média dos Elementos Medianos (MEM) tanto em tempo de execução como em quantidade de particionamentos executados. Isso é devido ao fato de que o algoritmo MEM necessita ter os registros ordenados dentro do *bucket*. Para tanto, ele precisa executar o método QuickSort quando precisa realizar o particionamento de um *bucket*, enquanto que o algoritmo do Centro de Massa não necessita desta ordenação interna de registros no *bucket*.

Como conclusão dos testes práticos, observa-se que as estratégias propostas para particionamento de blocos para o *Grid File* são mais escalonáveis e mais rápidas que a política tradicional de divisão, pois esta não consegue realizar eficientemente as divisões no *Grid File* quando é necessária a inserção de mais de 10.000 registros. Nesse caso, verifica-se um número muito grande de sobrecargas de *buckets* (*bucket overflow*). Assim, uma estrutura como a utilizada nos testes (com três dimensões, capacidade de três registros por *bucket* e escalas lineares que podem ter até 40 divisões em cada dimensão) não consegue suportar a maioria dos casos de inserção de registros usados nos testes. Isto pode ser observado nas colunas 40.000, 45.000, 50.000, 80.000 e 100.000 (que correspondem à quantidade de registros inseridos), conforme mostra a Tabela 5.5.

Capítulo 6

Conclusões e Trabalhos Futuros

Nessa dissertação foram propostas políticas de divisão para os *Grid Files*, onde a célula sobrecarregada da grade é particionada de acordo com pontos de distribuição uniforme dos registros da célula. Em outras palavras, as políticas propostas identificam o ponto que divide os registros contidos naquela grade uniformemente entre os dois novos *buckets* correspondentes à grade particionada. Para isto, foram desenvolvidos dois algoritmos. O Algoritmo da Média dos Elementos Medianos é um algoritmo que deve ser usado quando se requer uma distribuição exatamente uniforme. O Algoritmo do Centro de Massa é indicado como uma heurística para a solução do problema, pois seu resultado é aproximado, porém apresenta menor ordem de complexidade do que o primeiro, sendo indicado para situações em que uma aproximação do ponto de distribuição uniforme dos registros da célula é aceitável.

Este trabalho apresenta as seguintes contribuições na área de armazenamento e indexação de dados espaciais:

1. Redução da freqüência de particionamentos de células em estruturas de *Grid Files*, reduzindo assim o tempo de processamento global nas atualizações com inclusão de registros feitas em bancos de dados que tenham como base esta estrutura de dados, principalmente nos casos especiais onde o arquivo (base de dados) sofre constante aumento do número de registros (*growing file*) e, conseqüentemente, passa a ter

muitos *buckets* sobrecarregados (overflows).

2. A Revisão bibliográfica apresentada nesse trabalho pode servir de base para aprimorar o conhecimento de estruturas de arquivos com indexação e armazenamento através de múltiplas chaves de busca, como o próprio *Grid File*. Adicionalmente, pode funcionar como uma leitura complementar sobre o projeto e o funcionamento de Sistemas de Informação Geográficas (SIGs).

3. A aplicação das políticas propostas na construção de um protótipo de SIG usando o *Grid File*, como estratégia de armazenamento de dados espaciais.

A seguir são descritos trabalhos futuros que poderão dar continuidade ao que foi desenvolvido nesse trabalho de pesquisa.

6.1. Trabalhos Futuros

Propõe-se como um dos trabalhos futuros dessa pesquisa estender os testes sobre o sistema de *Grid Files*, para observar como ele se comporta diante de modelos de arquivos estáticos (*steady-state file*) e arquivos em contração (*shrinking file*).

O próximo passo desse projeto de pesquisa é implementar o algoritmo implementado em Java como um módulo DataBlade do Informix, que ficará agregado ao sistema do banco de dados e poderá ser usado como rotina básica do mesmo.

Para isto, no Apêndice A desta dissertação, encontra-se o código-fonte integral do programa em Java com as rotinas que simulam o funcionamento de um sistema baseado em *Grid Files*.

É importante, porém, notar que o código deverá sofrer severas modificações para adaptar os procedimentos de entrada/saída com a interface de troca de dados internos (I/O) no Informix. Contudo, estas alterações não irão alterar os principais procedimentos ou módulos do sistema, já que o modelo de algoritmo adotado separa claramente os procedimentos de tratamento de entrada/saída daqueles que tratam do problema do partionamento de *Grid Files*.

Um outro passo a ser dado na complementação deste trabalho é implementar políticas de união para células que fiquem com sua ocupação abaixo de um limite inferior mínimo de acordo com a aplicação, pois assim, o espaço de memória ocupado pelo arquivo será otimizado a medida que registros sejam removidos.

Referências Bibliográficas

[1] C.-H. Ang e H. Samet., "Approximate average storage utilization of bucket methods with arbitrary fanout," Nordic Journal of Computing, 3(3):280-291, Outono de 1996.

[2] Guttman, A., "R-Trees: A dynamic index structure for spatial searching," Proc. ACM SIGMOD, pp. 47-57, June 1984.

[3] Hinrichs, K., Nievergelt, J., "The grid file: A data structure designed to support proximity queries on spatial objects," Proc. Workshop on Graph Theoretic Concepts in Computer Science (Osnabruck, 1983).

[4] Lomet, D., Salzberg, B., "Spatial database access methods," SIGMOD RECORD, pp. 6-15 (1991).

[5] Nievergelt, J., Hinterberger, H., Sevcik, K.C., "The grid file: An adaptable, symmetric, multikey file structure," ACM Trans. on Database Systems, 9, 1, pp. 582 - 598 (1984).

[6] Robinson, J.T., "The K-D-B-tree: A search structure for large multidimensional dynamic indexes," Proc. ACM SIGMOD, pp. 10-18 (1981).

[7] Settis, T., Roussopoulos, N., Faloutsos, C., "The R+-Tree: A dynamic index for multi-dimensional objects," Proc. 13 VLDB Conference, Brighton 1987.

[8] Tamminen, M.,"Efficient spatial access to a data base," Proc. ACM SIGMOD, 200-206, 1982.

[9] Evangelidis, G., Lomet, D., Salzberg, B., "The hB$^{\pi}$ -Tree: a multi-attribute index supporting concurrency, recovery and node consolidation," 1995.

[10] Litwin, W., Neimat, M., Schneider, D. A., "LH*- A scalable, distributed data structure," ACM Transactions on Database Systems, December 1996, Pages 480-525.

[11] Gaede, V., Günther, O., "Multidimensional access methods," ACM Computing Surveys, June 1998.

[12] Câmara, G., Casanova, M. A., Hemerly, A. S., Magalhães, G. C., Medeiros, C. M. B., "Anatomia de sistemas de informação geográfica," 10^a Escola de Computação, Instituto de Computação, UNICAMP 1996.

[13] Ooi, Beng Chin, "Efficient query processing in geographic information systems," Lecture Notes in Computer Science 209, Springer-Verlag (1990).

[14] Hinrichs, K., "Implementation of the grid file: Design concepts and experience." BIT 25, 569-592 (1985).

[15] Jain, Anil K., "Fundamentals of digital image processing," Prentice-Hall International Editions, 1989.

[16] Güting, Ralf Hartmut, "An introduction to spatial database systems," VLDB Journal 3, 357-399 (1994).

[17] van Eck, J., Uffer, M., "A presentation of System 9," Photogrammetry and Land Information Systems, pg. 139-178, março 1989.

[18] Abraham Silberschatz e Henry F. Korth, "Database Systems," Makron Books - McGraw Hill, 1999.

[19] Ramez Elmasri e Shamkant B. Navathe, "Fundamentals of database systems," Addison-Wesley Publishing Company, 1994.

[20] R. A. Finkel, J. L. Bentley: Quad Trees: A data structure for retrieval on composite keys. Acta informatica 4, 1-9 (1974).

[21] A. Klinger: Patterns and search statistics. In: J. S. Rustagi (ed): "Optimizing methods in statistics," Academic Press, New York (1971).

[22] J. L. Bentley: "Multidimensional binary search trees used for associative searching," Comm. ACM 18, 9, 509-517 (1975).

[23] Matsuyama, T,. Hao, L. V., Nagao, M., "A file organization for geographic information systems based on spatial proximity," Int. Journal Comp. Vision, Graphics, and Image Processing 26, 3, 303-318 (1984).

[24] Ousterhout, J. K., "Corner stiching: A data structuring technique for VLSI layout tools," IEEE Trans. on Comp. Aided Desing CAD-3, 1, 87-100 (1984).

[25] Tamminen, M.,"The extendible cell method for closest points problems," BIT 22, 27-41, 1982.

[26] H. Kriegel, B. Seeger: "PLOP-Hashing: A grid file without directory," IEEE 4th International Conf. on Data Engineering, L.A., California, 369-376, 1988.

[27] Fussell, D., Kedem, Z. M., Silberschatz, A., "Deadlock removal using partial rollback in database systems," Proc. ACM SIGMOD, 65-73, 1981.

[28] H. Samet, "The quadtree and related hierarchical data structures," ACM Comp. Surveys 16, 187-260 (1984).

[29] Abel, D. J., Smith, J. L., "A data structure and algorithm based on a linear key for a rectangle retrieval problem," International Journal of Comp. Vision, Graphics, and Image Processing 24, 1, 1-13 (1983).

[30] Rosenberg, J. B., "Geographical data structures compared: A study of data structures supporting region queries," IEEE Trans. on Comp. Aided Desing CAD-4, 1, 53-67 (1985).

[31] Gunther, O., "Efficient structures for geometric data management," Lecture Notes in Computer Science 337, Springer-Verlag (1988).

[32] D. Comer, "The ubiquitous B-Tree," ACM Comp. Surveys 11, 2, 121-137 (1979).

[33] Jomier, G., Manouvrier, M., Rukoz, M., "Storage and Management of Similar Images," Journal of the Brazilian Computer Society n.3, vol. 6, 13-25 (2000).

[34] S. Morehouse, "The ARC/INFO geografic information system," Computers and Geosciences: An international journal, 18(4): 435-443 (1992).

[35] Câmara, G., Souza, R.C.M., Freitas, U.M., Casanova, M.A., "SPRING: Processamento de Imagens e Dados Georeferenciados," In: V Simpósio Brasileiro de Computação Gráfica e Processamento de Imagens, Lindóia, 1992. Anais, São José dos Campos, INPE, pp.233-242 (1992).

[36] Câmara, G., Souza, R.C.M., Freitas, U.M., Paiva, J.A.C., "SPRING: Concepção, Evolução, Perspectivas," In: VII Simpósio Brasileiro de Sensoriamento Remoto, Curitiba, PA, 1993, Anais, São José dos Campos, INPE (1993).

[37] University of California, "*Postgres 4.1 Reference Manual*" (1993).

[38] Bancilhon, F., Delobel, C., Kanellakis, P., "Building an Object-Oriented System - the Story of O2," Morgan Kaufmann (1992).

[39] Brayner, Angelo R. A., Notas de aula, Disciplina Tópicos Especiais em Bancos de Dados, UFC, 2000.

[40] K.C. Sevcik and N. Koudas - *"Filter Trees for Managing Spatial Data over a Range of Size Granularities"*. Proc. VLDB 1996, Mumbai, Índia (Sept. 96), pp.16-27.

[41] B. Moon, A. Acharya, J. Saltz - *"Study of Scalable Declustering Algorithms for Parallel Grid Files"*. IPPS 1996.

Apêndice **A**

Código-fonte do Protótipo em Java para Testes

```
1 import java.util.*;

2 class Registro
3 {
4    int regId;
5    float x;
6    float y;
7    float z;
8 }

9 class Coordenadas
10 {
11    float minX;
12    float minY;
13    float minZ;
14    float maxX;
15    float maxY;
16    float maxZ;
17 }
```

```
18 class Pos
19 {
20    int x;
21    int y;
22    int z;
23 }

24 class Bucket
25 {
26    int bucId;
27    int num_reg;
28    int num_cel;
29    Registro[] registros;
30    Vector celulas;

31    public Bucket()
32    {
33       registros = new Registro[Grid.c];
34       celulas = new Vector();
35    }
36 }

37 public class Grid
38 {

39    public static int tam_test = 6;  /* Especifica o tamanho
40    do vetor de inteiros usado para testar o funcionamento do
```

```
41    quick-sort */
42    public static int k = 3;  /* informa o numero de
43    atributos usados como chave no grid file */
44    public static int c = 3;  /* informa a capacidade dos
45    buckets de armazenar registros */

46    Bucket[][][] grid = new Bucket[40][40][40]; /* "grid" eh
47    uma matriz que aponta para buckets */

48    float lsTempX[] = {0, 500, 1000, 1500, 1750, 1875, 2000};
49    // Escala linear representando os anos de 0 a 2000
50    float lsTempY[] = {1, 4, 6, 10, 14, 17, 20, 26}; //
51    Escala linear representando as letras do alfabeto (A=1,
52    D=4 etc)
53    float lsTempZ[] = {1, 2, 3, 4, 5, 6, 7}; // Escala linear
54    representando os dias da semana (Dom=1, Seg=2 etc)

55    FloatVector lsX = new FloatVector(lsTempX);
56    FloatVector lsY = new FloatVector(lsTempY);
57    FloatVector lsZ = new FloatVector(lsTempZ);

58    int sX = 7; /* Tamanho da escala linear (quantidade de
59    divisões) no eixo X */
60    int sY = 8; /* Tamanho da escala linear (quantidade de
61    divisões) no eixo Y */
62    int sZ = 7; /* Tamanho da escala linear (quantidade de
63    divisões) no eixo Z */
```

```
64    int i; /* Representa a posição de um elemento no eixo X
65    */
66    int j; /* Representa a posição de um elemento no eixo Y
67    */
68    int l; /* Representa a posição de um elemento no eixo Z
69    */

70    float[] X = new float[c]; /* Representa os valores das
71    coordenadas dos registros no bucket a ser particionado em
72    cada dimensão */
73    float[] Y = new float[c];
74    float[] Z = new float[c];

75    int[] id = new int[c]; /* Representa os identificadores
76    de cada registro no bucket a ser particionado */

77    int regId = 0; /* Inicializa o identificador de registros
78    a partir de 0 */
79    int bucId = 0; /* Inicializa o identificador de buckets a
80    partir de 0 */
81    int dim = 0; /* Especifica a dimensão atual: dim=0 => X,
82    dim=1 => Y, dim=2 => Z */
83    int contadorDivisoes = 0; /* Registra a quantidade de
84    divisoes (splittings) aplicados */

85    static int inseriu = 0; /* Flag para indicar se algum
86    registro foi inserido na grid: inseriu=0 => não, inseriu=
87    1 => sim */
```

```
88    void init_vectors()
89    {
90       lsX = new FloatVector(lsTempX);
91       lsY = new FloatVector(lsTempY);
92       lsZ = new FloatVector(lsTempZ);
93    }

94    Bucket criar_bucket(int coordX, int coordY, int coordZ)
95    /* Cria e retorna um novo bucket alocado */
96    {
97       Bucket alocado = new Bucket();
98       Pos posicao = new Pos();
99       alocado.bucId = bucId;
100       bucId++;
101       alocado.num_reg = 0;
102       alocado.num_cel = 1;
103       posicao.x = coordX;
104       posicao.y = coordY;
105       posicao.z = coordZ;
106       alocado.celulas.add(posicao);
107       return alocado;
108    } // criar_bucket()

109    void coordenada2posicao(float x, float y, float z) /* A
110    posição relativa a cada eixo eh retornada nas variáveis
111    globais i, j e l */
112    {
```

```
113       i=0;
114       j=0;
115       l=0;

116       if (x < lsX.getFloat(0)) /* Coordenada passada tem
117       valor menor que o limite inferior da escala linear */
118       {
119          i = -1; /* Flag negativo */
120       }
121       else if (x > lsX.getFloat(sX - 1)) /* Coordenada
122       passada tem valor maior que o limite superior da
123       escala linear */
124       {
125          i = -2; /* Flag negativo */
126       }
127       else if (x == lsX.getFloat(sX - 1)) /* Caso o valor
128       passado seja o mesmo do limite inferior da escala
129       linear */
130       {
131          i = (sX - 2);
132       }
133       else
134       {
135          while ((x > lsX.getFloat(i + 1)) && (i < (sX - 1)))
136          /* Procura o intervalo em que o valor passado se
137          encaixa na escala linear */
138          {
139             i++;
```

```
140         }
141      }

142      if (y < lsY.getFloat(0)) /* Coordenada passada tem
143      valor menor que o limite inferior da escala linear */
144      {
145         j = -1; /* Flag negativo */
146      }
147      else if (y > lsY.getFloat(sY - 1)) /* Coordenada
148      passada tem valor maior que o limite superior da
149      escala linear */
150      {
151         j = -2; /* Flag negativo */
152      }
153      else if (y == lsY.getFloat(sY - 1)) /* Caso o valor
154      passado seja o mesmo do limite inferior da escala
155      linear */
156      {
157         j = (sY - 2);
158      }
159      else
160      {
161         while ((y > lsY.getFloat(j + 1)) && (j < (sY - 1)))
162         /* Procura o intervalo em que o valor passado se
163         encaixa na escala linear */
164         {
165            j++;
166         }
```

```
167       }

168       if (z < lsZ.getFloat(0)) /* Coordenada passada tem
169       valor menor que o limite inferior da escala linear */
170       {
171          l = -1; /* Flag negativo */
172       }
173       else if (z > lsZ.getFloat(sZ - 1)) /* Coordenada
174       passada tem valor maior que o limite superior da
175       escala linear */
176       {
177          l = -2; /* Flag negativo */
178       }
179       else if (z == lsZ.getFloat(sZ - 1)) /* Caso o valor
180       passado seja o mesmo do limite inferior da escala
181       linear */
182       {
183          l = (sZ - 2);
184       }
185       else
186       {
187          while ((z > lsZ.getFloat(l + 1)) && (l < (sZ - 1)))
188          /* Procura o intervalo em que o valor passado se
189          encaixa na escala linear */
190          {
191             l++;
192          }
193       }
```

```
194    } // coordenada2posicao()

195    Coordenadas posicao2coordenadas(Pos posicao) /* As
196    coordenadas relativas a cada eixo são retornadas na forma
197    de tipo de retorno de acordo com a posição da grid
198    passada*/
199    {
200       Coordenadas coord = new Coordenadas();
201       coord.minX = lsX.getFloat(posicao.x);
202       coord.maxX = lsX.getFloat(posicao.x + 1);
203       coord.minY = lsY.getFloat(posicao.y);
204       coord.maxY = lsY.getFloat(posicao.y + 1);
205       coord.minZ = lsZ.getFloat(posicao.z);
206       coord.maxZ = lsZ.getFloat(posicao.z + 1);
207       return coord;
208    } // posicao2coordenadas()

209    Bucket localizar_bucket(float x, float y, float z)
210    /* Recupera o bucket que contehm o registro nas
211    coordenadas x, y e z */
212    {
213       Bucket new_buc;
214       coordenada2posicao(x, y, z);

215       if ((i < 0) || (j < 0) || (l < 0)) /* Não existe
216       celula nem bucket correspondente */
217          {
218             System.out.print("** Erro: Não existe espaço
```

```
219             para armazenar o registro! ** \n");
220             System.exit(0);
221          }
222       if (grid[i][j][l] == null) /* Se o bucket ainda não
223       existir, ele eh criado */
224       {
225          new_buc = criar_bucket(i,j,l);
226          grid[i][j][l] = new_buc;
227       }
228       return grid[i][j][l];
229    } // localizar_bucket()

230    Registro recuperar_registro(Bucket b, int rid)
231    {
232       int aux = 0;
233       while ((b.registros[aux].regId != rid) && (aux <
234       b.num_reg))
235       {
236          aux++;
237       }
238       return b.registros[aux];
239    } // recuperar_registro()

240    void realocar_bucket(Bucket b1, Bucket b2, float ponto,
241    int opcao) /* Faz a distribuição (realocação) dos
242    registros dentro de seus respectivos buckets, depois
243    do particionamento.*/
244    {
```

```
245       Registro reg_temp;
246       Registro[] vet_reg1 = new Registro[c];
247       Registro[] vet_reg2 = new Registro[c];
248       int aux1, aux2 = 0, aux3;
249       int ind = 0, cont;
250       if (opcao == 1)
251       {
252          switch (dim)
253          {
254             case 0:  while (ponto > X[ind + 1])
255                      {
256                         ind++;
257                      }
258                      break;
259             case 1:  while (ponto > Y[ind + 1])
260                      {
261                         ind++;
262                      }
263                      break;
264             case 2:  while (ponto > Z[ind + 1])
265                      {
266                         ind++;
267                      }
268                      break;
269          }
270          if (b1.num_reg == Grid.c || b2.num_reg == Grid.c)
271          {
272             for (aux1 = 0; aux1 <= ind; aux1++)
```

```
273             {
274                reg_temp = recuperar_registro(b1, id[aux1]);
275                vet_reg1[aux1] = reg_temp;
276             }

277             for (aux3 = (ind + 1); aux3 < b1.num_reg;
278             aux3++)
279             {
280                reg_temp = recuperar_registro(b1, id[aux3]);
281                vet_reg2[aux2] = reg_temp;
282                aux2++;
283             }

284             for (aux3 = 0; aux3 <= aux1; aux3++)
285             {
286                b1.registros[aux3] = vet_reg1[aux3];
287             }

288             b1.num_reg = (aux1);

289             for (aux3 = 0; aux3 <= aux2; aux3++)
290             {
291                b2.registros[aux3] = vet_reg2[aux3];
292             }

293             b2.num_reg = (aux2);

294          }
```

```
295       }
296       else
297       {
298          int cont1=0, cont2=0;
299          switch (dim)
300          {
301             case 0:  for (cont=0; cont < b1.num_reg; cont++)
302                      {
303                         if (b1.registros[cont].x < ponto)
304                         {
305                            vet_reg1[cont1] =
306                            b1.registros[cont];
307                            cont1++;
308                         }
309                         else
310                         {
311                            vet_reg2[cont2] =
312                            b1.registros[cont];
313                            cont2++;
314                         }
315                      }
316                      for (cont=0; cont < b2.num_reg; cont++)
317                      {
318                         if (b2.registros[cont].x < ponto)
319                         {
320                            vet_reg1[cont1] =
321                            b2.registros[cont];
322                            cont1++;
```

```
323                         }
324                         else
325                         {
326                            vet_reg2[cont2] =
327                            b2.registros[cont];
328                            cont2++;
329                         }
330                      }
331                      break;
332            case 1:   for (cont=0; cont < b1.num_reg; cont++)
333                      {
334                         if (b1.registros[cont].y < ponto)
335                         {
336                            vet_reg1[cont1] =
337                            b1.registros[cont];
338                            cont1++;
339                         }
340                         else
341                         {
342                            vet_reg2[cont2] =
343                            b1.registros[cont];
344                            cont2++;
345                         }
346                      }
347                      for (cont=0; cont < b2.num_reg; cont++)
348                      {
349                         if (b2.registros[cont].y < ponto)
350                         {
```

```
351                         vet_reg1[cont1] =
352                         b2.registros[cont];
353                         cont1++;
354                      }
355                      else
356                      {
357                         vet_reg2[cont2] =
358                         b2.registros[cont];
359                         cont2++;
360                      }
361                   }
362                   break;
363            case 2:  for (cont=0; cont < b1.num_reg; cont++)
364                   {
365                      if (b1.registros[cont].z < ponto)
366                      {
367                         vet_reg1[cont1] =
368                         b1.registros[cont];
369                         cont1++;
370                      }
371                      else
372                      {
373                         vet_reg2[cont2] =
374                         b1.registros[cont];
375                         cont2++;
376                      }
377                   }
378                   for (cont=0; cont < b2.num_reg; cont++)
```

```
379                     {
380                        if (b2.registros[cont].z < ponto)
381                        {
382                           vet_reg1[cont1] =
383                           b2.registros[cont];
384                           cont1++;
385                        }
386                        else
387                        {
388                           vet_reg2[cont2] =
389                           b2.registros[cont];
390                           cont2++;
391                        }
392                     }
393                     break;
394           }

395           for (cont=0; cont < cont1; cont++)
396           {
397              b1.registros[cont] = vet_reg1[cont];
398           }
399           b1.num_reg = cont1;

400           for (cont=0; cont < cont2; cont++)
401           {
402              b2.registros[cont] = vet_reg2[cont];
403           }
404           b2.num_reg = cont2;
```

```
405       }

406    } // realocar_bucket()

407    void particionar_grid(float ponto, Bucket bucket, int
408    opcao) /* eh passado qual o ponto a ser feito o
409    particionamento*/
410    {
411       /* Neste procedimento deve-se dividir todo o grid file
412       de acordo com a
413       dimensão e a posição passadas como parâmetro,
414       realocando os registros */

415       int m, n, o;
416       Bucket new_buc, buc;
417       Pos position = new Pos();

418       switch (dim)
419       {
420          case 0: /* Ajustar o particionamento do espaço da
421          grid na escala
422                linear correspondente (dim=0 => Eixo X) */

423 //             INÍCIO do particionamento e ajuste da escala
424 linear.

425                try
426                {
```

```
427                     lsX.add(sX, lsX.getFloat(sX - 1));
428                     for (m = (sX - 1); m > (i + 1); m--)
429                     {
430                        lsX.set(m, lsX.getFloat(m - 1));
431                     }
432                     lsX.set((i + 1), ponto);
433                     sX++;
434                  }
435                  catch (Exception exception)
436                  {
437                     System.out.print("ERRO no particionamento
438                     e ajuste da escala linear em X\n");
439                     System.exit(0);
440                  }

441 //               FIM do particionamento e ajuste da escala
442 linear.

443 //               INÍCIO do particionamento e ajuste do grid
444 directory.

445                  for (m = (sX - 1); m > i; m--)
446                  {
447                     for (n = 0; n < sY; n++)
448                     {
449                        for (o = 0; o < sZ; o++)
450                        {
451                           try
```

```
452                     {
453                        grid[m][n][o] = grid[m-1][n][o];
454                     }
455                     catch (Exception excecao)
456                     {
457                        System.out.print("ERRO em grid[m]
458                        [n][o] = grid[m-1][n][o] => case
459                        0: Os valores sao m="+m+",
460                        n="+n+" e o="+o+"!\nO Erro foi
461                        "+excecao+"\n");
462                        System.exit(0);
463                     }

464                     if ((m == (i+1)) && (n != j) && (o
465                     != l))
466                     {
467                        position.x = m;
468                        position.y = n;
469                        position.z = o;
470                        buc = grid[m][n][o];
471                        if (buc == null)
472                        {
473                           try
474                           {
475                              buc = criar_bucket(m, n,
476                              o);
477                           }
478                           catch (Exception excecao)
```

```
479                                 {
480                                    System.out.print("ERRO em
481                                    buc = criar_bucket(m, n,
482                                    o)! O Erro foi
483                                    "+excecao+"\n");
484                                    System.exit(0);
485                                 }
486                              }
487                              try
488                              {
489                                 buc.celulas.add(position);
490                              }
491                              catch (Exception e)
492                              {
493                                 System.out.print("ERRO em:
494                                 buc.celulas[buc.num_cel] =
495                                 position. \nErro: "+e+"\n");
496                                 System.exit(0);
497                              }
498                              try
499                              {
500                                 buc.num_cel++;
501                              }
502                              catch (Exception e)
503                              {
504                                 System.out.print("ERRO em:
505                                 buc.num_cel++;\n");
506                                 System.exit(0);
```

```
507                         }
508                         try
509                         {
510                            grid[m][n][o] = buc;
511                         }
512                         catch (Exception e)
513                         {
514                            System.out.print("ERRO em:
515                            grid[m][n][o] = buc;\n");
516                            System.exit(0);
517                         }
518                      }
519                   }
520                }
521             }

522             new_buc = criar_bucket(i+1, j, l);
523             try
524             {
525                grid[i+1][j][l] = new_buc;
526             }
527             catch (Exception e1)
528             {
529                System.out.print("ERRO na atribuição:
530                grid[i+1][j][l] = new_buc;\n");
531                System.exit(0);
532             }
533             if (grid[i][j][l] == null)
```

```
534                {
535                  //System.out.print("O bucket correspondente
536                  a posicao i,j,l do grid eh NULO!\n");
537                }
538                try
539                {
540                   realocar_bucket(grid[i][j][l], grid[i+1]
541                   [j][l], ponto, opcao);
542                }
543                catch (Exception e2)
544                {
545                   System.out.print("ERRO na chamada a
546                   realocar bucket no case 0\n");
547                   System.exit(0);
548                }
549                break;

550          case 1: /* Ajustar o particionamento do espaço da
551          grid na escala
552                linear correspondente (dim=1 => Eixo Y) */

553                try
554                {
555                   lsY.add(sY, lsY.getFloat(sY - 1));
556                   for (n = sY; n > (j + 1); n--)
557                   {
558                      lsY.set(n, lsY.getFloat(n - 1));
559                   }
```

```
560                  lsY.set((j + 1), ponto);
561                  sY++;
562               }
563               catch (Exception exception)
564               {
565                  System.out.print("ERRO no particionamento
566                  e ajuste da escala linear em Y\n");
567                  System.exit(0);
568               }

569               for (m = 0; m < sX; m++)
570               {
571                  for (n = (sY - 1); n > j; n--)
572                  {
573                     for (o = 0; o < sZ; o++)
574                     {
575                        try
576                        {
577                           grid[m][n][o] = grid[m][n-1][o];
578                        }
579                        catch (Exception excecao)
580                        {
581                           System.out.print("ERRO em grid[m]
582                           [n][o] = grid[m][n-1][o] => case
583                           1: Os valores sao m="+m+",
584                           n="+n+" e o="+o+"!\nO Erro foi
585                           "+excecao+"\n");
586                           System.exit(0);
```

```
587                         }
588                         if ((m != i) && (n == (j + 1)) && (o
589                         != l))
590                         {
591                            position.x = m;
592                            position.y = n;
593                            position.z = o;
594                            buc = grid[m][n][o];
595                            if (buc == null)
596                            {
597                               try
598                               {
599                                  buc = criar_bucket(m, n,
600                                  o);
601                               }
602                               catch (Exception excecao)
603                               {
604                                  System.out.print("ERRO em
605                                  buc = criar_bucket(m, n,
606                                  o)! O Erro foi
607                                  "+excecao+"\n");
608                                  System.exit(0);
609                               }
610                            }
611                            try
612                            {
613                               buc.celulas.add(position);
614                            }
```

```
615                         catch (Exception e)
616                         {
617                            System.out.print("ERRO em
618                            buc.celulas[buc.num_cel] =
619                            position. \nErro: "+e+"\n");
620                            System.exit(0);
621                         }
622                         try
623                         {
624                            buc.num_cel++;
625                         }
626                         catch (Exception e)
627                         {
628                            System.out.print("ERRO em
629                            buc.num_cel++;\n");
630                            System.exit(0);
631                         }
632                         try
633                         {
634                            grid[m][n][o] = buc;
635                         }
636                         catch (Exception e)
637                         {
638                            System.out.print("ERRO em
639                            grid[m][n][o] = buc;\n");
640                            System.exit(0);
641                         }
642                      }
```

```
643                     }
644                  }
645               }

646               new_buc = criar_bucket(i, j+1, l);
647               try
648               {
649                  grid[i][j+1][l] = new_buc;
650               }
651               catch (Exception e1)
652               {
653                  System.out.print("ERRO na atribuição:
654                  grid[i][j+1][l] = new_buc;\n");
655                  System.exit(0);
656               }
657               if (grid[i][j][l] == null)
658               {
659                 //System.out.print("O bucket correspondente
660                 a posicao i,j,l do grid eh NULO!\n");
661               }
662               try
663               {
664                  realocar_bucket(grid[i][j][l], grid[i]
665                  [j+1][l], ponto, opcao);
666               }
667               catch (Exception e2)
668               {
669                  System.out.print("ERRO na chamada a
```

```
670                realocar bucket no case 1\n");
671                System.exit(0);
672             }
673             break;

674       case 2: /* Ajustar o particionamento do espaço da
675       grid na escala
676             linear correspondente (dim=2 => Eixo Z) */

677             try
678             {
679                lsZ.add(sZ, lsZ.getFloat(sZ - 1));
680                for (o = sZ; o > (l + 1); o--)
681                {
682                   lsZ.set(o, lsZ.getFloat(o - 1));
683                }
684                lsZ.set((l + 1), ponto);
685                sZ++;
686             }
687             catch (Exception exception)
688             {
689                System.out.print("ERRO no particionamento
690                e ajuste da escala linear em Z\n");
691                System.exit(0);
692             }

693             for (m = 0; m < sX; m++)
694             {
```

```
695                     for (n = 0; n < sY; n++)
696                     {
697                        for (o = (sZ - 1); o > l; o--)
698                        {
699                           try
700                           {
701                              grid[m][n][o] = grid[m][n][o-1];
702                           }
703                           catch (Exception excecao)
704                           {
705                              System.out.print("ERRO em grid[m]
706                              [n][o] = grid[m][n][o-1] => case
707                              2: Os valores sao m="+m+",
708                              n="+n+" e o="+o+"!\nO Erro foi
709                              "+excecao+"\n");
710                              System.exit(0);
711                           }
712                           if ((m != i) && (n != j) && (o == (l
713                           + 1)))
714                           {
715                              position.x = m;
716                              position.y = n;
717                              position.z = o;
718                              buc = grid[m][n][o];
719                              if (buc == null)
720                              {
721                                 try
722                                 {
```

```
723                         buc = criar_bucket(m, n,
724                         o);
725                      }
726                      catch (Exception excecao)
727                      {
728                         System.out.print("ERRO em
729                         buc = criar_bucket(m, n,
730                         o)! O Erro foi
731                         "+excecao+"\n");
732                         System.exit(0);
733                      }
734                   }
735                   try
736                   {
737                      buc.celulas.add(position);
738                   }
739                   catch (Exception e)
740                   {
741                      System.out.print("ERRO em:
742                      buc.celulas.add(position).
743                      \nErro: "+e+"\n");
744                      System.exit(0);
745                   }
746                   try
747                   {
748                      buc.num_cel++;
749                   }
750                   catch (Exception e)
```

```
751                         {
752                            System.out.print("ERRO em:
753                            buc.num_cel++\n");
754                            System.exit(0);
755                         }
756                         try
757                         {
758                            grid[m][n][o] = buc;
759                         }
760                         catch (Exception e)
761                         {
762                            System.out.print("ERRO em:
763                            grid[m][n][o] = buc;\n");
764                            System.exit(0);
765                         }
766                      }
767                   }
768                }
769             }

770             new_buc = criar_bucket(i, j, l+1);
771             try
772             {
773                grid[i][j][l+1] = new_buc;
774             }
775             catch (Exception e1)
776             {
777                System.out.print("ERRO na atribuição:
```

```
778                    grid[i][j][l+1] = new_buc;\n");
779                    System.exit(0);
780                }
781                if (grid[i][j][l] == null)
782                {
783                  //System.out.print("O bucket correspondente
784                  a posicao i,j,l do grid eh NULO!\n");
785                }
786                try
787                {
788                   realocar_bucket(grid[i][j][l], grid[i][j]
789                   [l+1], ponto, opcao);
790                }
791                catch (Exception e2)
792                {
793                   System.out.print("ERRO na chamada a
794                   realocar bucket no case 2\n");
795                   System.exit(0);
796                }
797                break;

798          default: System.out.print("** Erro: Nao existe a
799          dimensao requerida para a operacao de
800          particionamento! ** \n");
801                   System.exit(0);
802       }

803    } // particionar_grid()
```

```
804    float calcula_media(float a[], int lim)
805    {
806       /* Este procedimento calcula e retorna a media
807       aritmehtica dos lim
808       primeiros elementos(reais) do vetor a[] de floats */

809       float soma = (float)0.0;
810       int cont;
811       for (cont = 0; cont < lim; cont++)
812       {
813          soma = a[cont] + soma;
814       }
815       return (float) soma/lim;
816    } // calcula_media()

817    Bucket dividir_bucket(Bucket buc, int opcao)
818    {
819       int cont, ind1, ind2, pos;
820       float ponto_medio = 0.0f;
821       for (cont=0; cont < buc.num_reg; cont++)
822       {
823          X[cont] = buc.registros[cont].x;
824          Y[cont] = buc.registros[cont].y;
825          Z[cont] = buc.registros[cont].z;
826          id[cont] = buc.registros[cont].regId;
827       }
```

```
828       /* Escolher a dimensão a ser particionada de acordo
829       com o mehtodo FIFO */

830       if (dim < 2) /* Se a dimensão atual for X(dim=0),
831       então recebe Y(dim=1)
832                   Se a dimensão atual for Y(dim=1), então
833                   recebe Z(dim=2)
834                   Se a dimensão atual for Z(dim=2), então
835                   recebe X(dim=0) */
836       {
837          dim++;
838       }
839       else
840       {
841          dim = 0;
842       }

843       if (opcao == 1) /* O algoritmo escolhido foi Media dos
844       Elementos Centrais */
845       {

846          /* Usar o metodo Quick-Sort para ordenar os
847          elementos (registros)
848             dentro de um bucket específico */

849          switch (dim)
850          {
851             case 0:
```

```
852                     quicksort(X, 0, buc.num_reg - 1);
853                 break;
854             case 1:
855                     quicksort(Y, 0, buc.num_reg - 1);
856                 break;
857             case 2:
858                     quicksort(Z, 0, buc.num_reg - 1);
859                 break;
860         }
861         /* Calcula os índices dos dois registros medianos
862         do bucket em relação
863         a coordenada escolhida para o particionamento */

864         ind1 = (int) (buc.num_reg / 2);
865 /*
866         if ((buc.num_reg % 2) > 0) // Se a divisão não for
867         exata, ocorre o arredondamento para cima
868         {
869             ind1++;
870         }
871 */
872         ind2 = ind1 + 1;

873         switch (dim)
874         {
875             case 0: ponto_medio = (float) ((X[ind1] +
876             X[ind2]) / 2);
877                 break;
```

```
878             case 1: ponto_medio = (float) ((Y[ind1] +
879             Y[ind2]) / 2);
880                   break;
881             case 2: ponto_medio = (float) ((Z[ind1] +
882             Z[ind2]) / 2);
883                   break;
884          }
885       }
886       else if (opcao == 2) /* O algoritmo escolhido foi o do
887       Centro de Massa */
888       {
889          switch (dim)
890          {
891             case 0: ponto_medio = calcula_media(X,
892             buc.num_reg);
893                   break;
894             case 1: ponto_medio = calcula_media(Y,
895             buc.num_reg);
896                   break;
897             case 2: ponto_medio = calcula_media(Z,
898             buc.num_reg);
899                   break;
900          }
901       }
902       else if (opcao == 3) /* O algoritmo escolhido foi o de
903       divisão tradicional */
904       {
905          Coordenadas coord = posicao2coordenadas((Pos)
```

```
906          buc.celulas.get(0));
907          switch (dim)
908          {
909             case 0: ponto_medio = (coord.minX + coord.maxX)
910             / 2;
911                   break;
912             case 1: ponto_medio = (coord.minY + coord.maxY)
913             / 2;
914                   break;
915             case 2: ponto_medio = (coord.minZ + coord.maxZ)
916             / 2;
917                   break;
918          }
919       }
920       contadorDivisoes++;
921       particionar_grid(ponto_medio, buc, opcao);
922       return buc;
923    } // dividir_bucket()

924    void quicksort(float a[], int lo0, int hi0)
925    {
926       float pivot;
927       int pivaux;
928       int lo = lo0;
929       int hi = hi0;
930       if (lo >= hi)
931       {
932          return;
```

```
933       }
934       else if(lo == (hi-1))
935       { /* ordena uma lista de dois elementos trocando-os se
936       necessário */
937          if (a[lo] > a[hi])
938          {
939             float T = a[lo];
940             int V = id[lo];
941             a[lo] = a[hi];
942             id[lo] = id[hi];
943             a[hi] = T;
944             id[hi] = V;
945          }
946          return;
947       }
948       /*  Escolhe um pivô e move-o para fora do caminho */

949       pivot = a[(lo + hi) / 2];
950       pivaux = id[(lo + hi) / 2];
951       a[(lo + hi) / 2] = a[hi];
952       id[(lo + hi) / 2] = id[hi];
953       a[hi] = pivot;
954       id[hi] = pivaux;

955       while( lo < hi )
956       { /* Busca para a frente de a[lo] ateh ser encontrado
957       um elemento que
958           eh maior do que o pivo ou lo >= hi */
```

```
959           while (a[lo] <= pivot && lo < hi)
960           {
961             lo++;
962           }

963         /* Busca retrocesso de a[hi] ateh ser encontrado um
964         elemento que eh
965           menor de que o pivô, ou lo >= hi */
966           while (pivot <= a[hi] && lo < hi )
967           {
968             hi--;
969           }

970           /* Troca os elementos a[lo] e a[hi] */
971           if(lo < hi)
972           {
973             float T = a[lo];
974             int V = id[lo];
975             a[lo] = a[hi];
976             id[lo] = id[hi];
977             a[hi] = T;
978             id[hi] = V;
979           }
980       }

981       /* Coloca o mediano no "centro" da lista */
982       a[hi0] = a[hi];
983       id[hi0] = id[hi];
```

```
984       a[hi] = pivot; /* Chamadas recursivas, elementos
985       a[lo0] ateh a[lo-1] são
986                       menores do que ou iguais ao pivô,
987                       elementos a[hi+1]
988                       ateh a[hi0] são maiores do que o pivô.
989                       */
990       id[hi] = pivaux;
991       quicksort(a, lo0, lo-1);
992       quicksort(a, hi+1, hi0);
993    } // quicksort()

994    Registro novo_registro(float x, float y, float z)
995    /* Retorna um novo registro com os dados passados como
996    parâmetros */
997    {
998       Registro reg = new Registro();
999       reg.regId = regId;
1000       regId++;
1001       reg.x = x;
1002       reg.y = y;
1003       reg.z = z;
1004       return reg;
1005    } // novo_registro()

1006    Bucket inserir_registro(Bucket b, Registro r, int
1007    opDefault)
1008    {
1009       int op = 0;
```

```
1010        if (b.num_reg < c)
1011        {
1012           b.registros[b.num_reg] = r;
1013           b.num_reg++;
1014        }
1015        else if (b.num_cel > 1)
1016        {
1017           /* Dividir a região do bucket entre as várias
1018           celulas que ela cobre */

1019 /*
1020           int half = (int) b.num_cel/2;
1021           int cont = 0;
1022           Pos posicao = (Pos) b.celulas.get(half);
1023           Bucket new_buc = criar_bucket(posicao.x, posicao.y,
1024           posicao.z);
1025           for (int i = (half+1); i < b.num_cel; i++)
1026           {
1027              new_buc.celulas.add(b.celulas.get(i));
1028              new_buc.num_cel++;
1029           }
1030           for (int i = (b.num_cel - 1); i >= half; i--)
1031           {
1032              b.celulas.remove(i);
1033              cont++;
1034           }
1035           b.num_cel = (b.num_cel - cont);
1036 */
```

```
1037          // Inserir o registro a ser inserido no devido
1038          bucket e particionar os registros entre os dois
1039          buckets gerados.

1040       }
1041       else
1042       {
1043          if (opDefault == 0)
1044          {
1045             /* Caso o bucket sobrecarregado corresponda a
1046             uma unica celula,
1047                disparar o splitting de bucket */
1048             System.out.print("Aviso: O bucket correspondente
1049             está com a capacidade esgotada! \n");
1050             System.out.print("Escolha qual algoritmo de
1051             splitting deve ser usado: \n");
1052             System.out.print("1 - Media dos Elementos
1053             Centrais \n");
1054             System.out.print("2 - Centro de Massa \n");
1055             System.out.print("3 - Divisão Tradicional \n");
1056             System.out.print("Opção: ");
1057             op = Util.readInt();
1058          }
1059          else
1060          {
1061             op = opDefault;
1062          }
```

```
1063          grid[i][j][l] = b;
1064          b = dividir_bucket(b, op);

1065          if (b.num_reg < c)
1066          {
1067             b.registros[b.num_reg] = r;
1068             b.num_reg++;
1069          }
1070       }
1071       return b;
1072    } // inserir_registro()

1073    void printHeader()
1074    {
1075       System.out.print("    ** SISTEMA DE SIMULACAO DE
1076       FUNCIONAMENTO DE ESTRUTURAS DE GRID FILES ** \n");
1077       System.out.print("\n");
1078       System.out.print("Este sistema esta simulando um Grid
1079       File com "+k+" dimensoes. \n");
1080       System.out.print("O sistema esta configurado para
1081       aceitar no maximo "+c+" registro(s) por bucket. \n");
1082       System.out.print("\n");
1083       System.out.print("O Eixo X constitui uma escala linear
1084       representando os anos de 0 a 2000 \n");
1085       System.out.print("distribuídos da seguinte forma:
1086       \n");
1087       System.out.print(lsX.getFloat(0)+", "+lsX.getFloat(1)
1088       +", "+lsX.getFloat(2)+", "+lsX.getFloat(3)+",
```

```
1089        "+lsX.getFloat(4)+", "+lsX.getFloat(5)+",
1090        "+lsX.getFloat(6)+" \n");
1091        System.out.print("O valor de sX eh "+sX+"\n");
1092        System.out.print("O Eixo Y constitui uma escala linear
1093        representando as letras do alfabeto \n");
1094        System.out.print("distribuídas da seguinte forma:
1095        \n");
1096        System.out.print("A="+lsY.getFloat(0)+",
1097        D="+lsY.getFloat(1)+", F="+lsY.getFloat(2)+",
1098        J="+lsY.getFloat(3)+", N="+lsY.getFloat(4)+",
1099        Q="+lsY.getFloat(5)+", T="+lsY.getFloat(6)+",
1100        Z="+lsY.getFloat(7)+" \n");
1101        System.out.print("O valor de sY eh "+sY+"\n");
1102        System.out.print("O Eixo Z constitui uma escala linear
1103        representando os dias da semana \n");
1104        System.out.print("distribuídos da seguinte forma:
1105        \n");
1106        System.out.print("Dom="+lsZ.getFloat(0)+",
1107        Seg="+lsZ.getFloat(1)+", Ter="+lsZ.getFloat(2)+",
1108        Qua="+lsZ.getFloat(3)+", Qui="+lsZ.getFloat(4)+",
1109        Sex="+lsZ.getFloat(5)+", Sab="+lsZ.getFloat(6)+" \n");
1110        System.out.print("O valor de sZ eh "+sZ+"\n");
1111    } // printHeader()

1112    void inserir()
1113    {
1114        char ch, ran;
1115        float x, y, z;
```

```
1116        Bucket buc;
1117        Registro reg;
1118        int quant, loop, algSplit;
1119        Date t1, t2;
1120        long time;
1121 /*
1122        System.out.print("Deseja inserir registros
1123        manualmente? (S/N) ");
1124        ch = Util.readChar();
1125        if (ch=='s'||ch=='S')
1126        {
1127           inseriu++; // Indica que um registro foi inserido
1128           no Grid File
1129           while (ch == 's' || ch == 'S')
1130           {
1131              System.out.print("Qual a posição no eixo X? ");
1132              x = (float) Util.readDbl();
1133              System.out.print("Qual a posição no eixo Y? ");
1134              y = (float) Util.readDbl();;
1135              System.out.print("Qual a posição no eixo Z? ");
1136              z = (float) Util.readDbl();;
1137              buc = localizar_bucket(x, y, z);
1138              reg = novo_registro(x, y, z);
1139              buc = inserir_registro(buc, reg, 0);
1140              grid[i][j][l] = buc;
1141              System.out.print("Deseja inserir um novo
1142              registro? (S/N) ");
1143              ch = Util.readChar();
```

```
1144          }
1145       }
1146       else
1147       {
1148 */
1149          System.out.print("Deseja usar a entrada automatica
1150          (randomica) de dados? (S/N) ");
1151          ran = Util.readChar();
1152          if (ran=='s' || ran=='S')
1153          {
1154             inseriu++; // Indica que um registro foi
1155             inserido no Grid File
1156             System.out.print("Quantos registros deseja
1157             gerar? ");
1158             quant = Util.readInt();;
1159             System.out.print("Escolha o algoritmo de divisao
1160             que deve ser usado se houver sobrecarga de
1161             bucket: \n");
1162             System.out.print("0 - Solicita escolha do
1163             usuario a cada sobrecarga \n");
1164             System.out.print("1 - Media dos Elementos
1165             Centrais \n");
1166             System.out.print("2 - Centro de Massa \n");
1167             System.out.print("3 - Divisao Tradicional \n");
1168             System.out.print("Opcao: ");
1169             algSplit = Util.readInt();
1170             t1 = new Date();
1171             Random random = new Random();
```

```
1172            for (loop = 0; loop < quant; loop++)
1173            {
1174               x = (float) geraRandom(2001, random);
1175               y = (float) (geraRandom(26, random) + 1);
1176               z = (float) (geraRandom(7, random) + 1);
1177               buc = localizar_bucket(x, y, z);
1178               reg = novo_registro(x, y, z);
1179               buc = inserir_registro(buc, reg, algSplit);
1180               grid[i][j][l] = buc;
1181            }
1182            t2 = new Date();
1183            time = t2.getTime() - t1.getTime();
1184            System.out.print("O tempo para realizar a
1185            operação foi de "+time+" milisegundos! \n");
1186            System.out.print("A quantidade de
1187            particionamentos executada foi de
1188            "+contadorDivisoes+" divisoes! \n");
1189          }

1190 //      }

1191       System.out.print("** Entrada de dados finalizada!
1192       **\n");
1193       System.out.print("\n");
1194    } // inserir()

1195    int geraRandom(int scope, Random random)
1196    {
```

```
1197       int gerado;
1198 //      Random random = new Random();
1199       gerado = random.nextInt();
1200       if (gerado < 0)
1201       {
1202          gerado *= -1;
1203       }
1204       gerado = (gerado % scope);
1205 //      random = null;
1206       return gerado;
1207    } // geraRandom()

1208    void exibir()
1209    {
1210       int cont;
1211       char ch;
1212       float x, y, z;
1213       Bucket buc;
1214       System.out.print("Deseja visualizar os dados de um
1215       bucket? (S/N) ");
1216       ch = Util.readChar();
1217       System.out.print("\n");
1218       while (ch == 's' || ch == 'S')
1219       {
1220          System.out.print("Qual a posicao no eixo X? ");
1221          x = (float) Util.readDbl();;
1222          System.out.print("Qual a posicao no eixo Y? ");
1223          y = (float) Util.readDbl();;
```

```
1224          System.out.print("Qual a posicao no eixo Z? ");
1225          z = (float) Util.readDbl();;
1226          System.out.print("\n");
1227          coordenada2posicao(x, y, z);
1228          if (grid[i][j][l] == null) /* Se o bucket ainda não
1229          existir */
1230          {
1231             System.out.print("O Bucket procurado nao existe
1232             ou ainda nao foi alocado!\n");
1233             System.out.print("\n");
1234          }
1235          else
1236          {
1237             buc = grid[i][j][l];
1238             System.out.print("<-- O Identificador do Bucket
1239             selecionado eh "+buc.bucId+". -->\n");
1240             System.out.print("O numero de registros eh
1241             "+buc.num_reg+". \n");
1242             if (buc.num_reg > 0)
1243             {
1244                System.out.print("\n");
1245                System.out.print("Os registros armazenados
1246                neste bucket são: \n");
1247                for (cont = 0; cont < buc.num_reg; cont++)
1248                {
1249                   System.out.print("O identificador do
1250                   registro eh "+buc.registros[cont].regId+".
1251                   \n");
```

```
1252                    System.out.print("O valor do eixo X eh
1253                    "+buc.registros[cont].x+". \n");
1254                    System.out.print("O valor do eixo Y eh
1255                    "+buc.registros[cont].y+". \n");
1256                    System.out.print("O valor do eixo Z eh
1257                    "+buc.registros[cont].z+". \n");
1258                 }
1259                 System.out.print("<-- Fim dos registros do
1260                 Bucket de Identificador: "+buc.bucId+" -->
1261                 \n");
1262                 System.out.print("\n");
1263              }
1264              else
1265                 System.out.print("O Bucket selecionado esta
1266                 vazio. \n");
1267           }
1268           System.out.print("Deseja visualizar um outro
1269           bucket? (S/N) ");
1270           ch = Util.readChar();
1271           System.out.print("\n");
1272        }
1273        System.out.print("** Saida de dados finalizada!
1274        **\n");
1275        System.out.print("\n");
1276     } // exibir()

1277     void footNote()
1278     {
```

```
1279        System.out.print("   ** O SISTEMA DE SIMULACAO DE GRID
1280        FILES FOI FINALIZADO COM SUCESSO ** \n");
1281        System.out.print("\n");
1282        System.out.print("Tecle <ENTER> para sair...");
1283        Util.readEnter();
1284     } // footNote()

1285     public static void main(String argv[])
1286     {
1287 //      clr();
1288 //      randomize();
1289        Grid g = new Grid();
1290        g.printHeader();
1291 //       g.init_vectors();
1292        g.inserir();
1293        if (g.inseriu > 0)
1294        {
1295           g.exibir();
1296        }
1297        g.footNote();
1298     }

1299 }
1300 // FIM
```