# Fatores que Afetam o Desempenho de Métodos de Junções Espaciais: um Estudo Baseado em Dados Reais

*Sandro Danilo Gatti*

**Dissertação de Mestrado**

Instituto de Computação
Universidade Estadual de Campinas

# Fatores que Afetam o Desempenho de Métodos de Junções Espaciais: um Estudo Baseado em Dados Reais

**Sandro Danilo Gatti**

Junho de 2000

**Banca Examinadora:**

- Prof. Dr. Geovane Cayres Magalhães (Orientador)
- Profa. Dra. Ana Carolina Salgado
  Centro de Informática – Universidade Federal de Pernambuco
- Profa. Dra. Claudia Bauzer Medeiros
  Instituto de Computação – Universidade Estadual de Campinas
- Prof. Dr. Neucimar Jerônimo Leite (Suplente)
  Instituto de Computação – Universidade Estadual de Campinas

Gatti, Sandro Danilo

G229f Fatores que afetam o desempenho de métodos de junções espaciais: um estudo com base em dados reais / Sandro Danilo Gatti – Campinas, [S.P. :s.n.], 2000.

Orientador : Geovane Cayres Magalhães

Dissertação (mestrado) - Universidade Estadual de Campinas, Instituto de Computação.

1. Sistema de informação geográfica. 2. Estrutura de dados (Computação). 3. Banco de dados - Gerência. I. Magalhães, Geovane Cayres. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

# Fatores que Afetam o Desempenho de Métodos de Junções Espaciais: um Estudo Baseado em Dados Reais

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Sandro Danilo Gatti e aprovada pela Banca Examinadora.

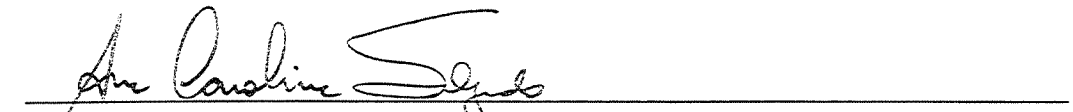
Campinas, 25 de junho de 2000.

Prof. Dr. Geovane Cayres Magalhães
(Orientador)

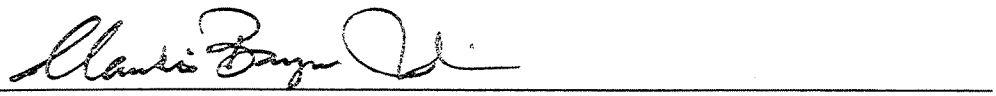
Dissertação apresentada ao Instituto de Computação, UNICAMP, como requisito parcial para a obtenção do título de Mestre em Ciência da Computação.

# TERMO DE APROVAÇÃO

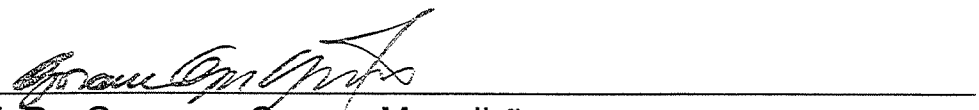
Tese defendida e aprovada em 31 de julho de 2000, pela Banca Examinadora composta pelos Professores Doutores:



Profa. Dra. Ana Carolina Salgado
UFPE – CI



Profa. Dra. Cláudia M. Bauzer Medeiros
IC – UNICAMP



Prof. Dr. Geovane Cayres Magalhães
IC – UNICAMP

A Célio, Lidernei e Leony
por tudo aquilo que são...

Senhor, fazei com que procure mais
consolar, que ser consolado,
compreender que ser compreendido,
amar que ser amado.
Pois é dando que se recebe,
é perdoando que se é perdoado,
e é morrendo que se vive para a vida eterna.

*Oração de São Francisco*

# Agradecimentos

Agradeço inicialmente e principalmente a Deus, pois tudo o que somos devemos a Ele. Agradeço pelo seu amor, seus dons, sua força, as graças a mim ofertadas.

Aos meus pais, Célio e Lidernei, pois me deram a vida, deram seu amor, sua dedicação, seu apoio.

À minha esposa, Leony, e aos meus irmãos, Charles e Danúsia, por sua compreensão, seu apoio, os momentos de alegria.

Aos muitos amigos que fiz aqui: Hélio, Guilherme, Kelcy, Emílio, Elisângela, Alexandre Pereira, Gianfrancesca, Alexandre Braga, Cesar, Daniel Kaster, e tantos outros, que seria necessário mais de uma simples página para enumerá-los. E aos que trouxe comigo da graduação e que caminharam ao meu lado durante este trabalho. Obrigado por sua amizade e desculpe os momentos de mau-humor.

Ao pessoal do Grupo de Banco de Dados, em especial à professora Claudia. Ao meu orientador, Geovane. Obrigado por partilhar sua experiência, sua vivência.

Ao CNPq e ao MCT Pronex II/SAI, pelo suporte financeiro a este trabalho. Ao Instituto de Computação, que me deu a oportunidade.

E a todos os que, por ventura, tenha me esquecido de colocar aqui. Meus mais sinceros agradecimentos.

# Resumo

Métodos de junção espacial baseadas em percurso sincronizado em árvores foram analizadas. Os fatores considerados incluíram tamanho do *bufferpool*, tamanho da página, critérios de ordenação de índices intermediários de junção, políticas de substituição de páginas do *bufferpool*, dentre outros fatores. Esta análise foi efetuada tomando-se por base um conjunto de dados reais extraídos de uma aplicação de SIG em telecomunicações, indexados em R*-trees. Os resultados deste trabalho avaliam a forma como estes fatores afetam o desempenho das junções espaciais e podem ser utilizados para ajustar estas técnicas.

# Abstract

Synchronized tree traversal join methods for spatial access methods were analysed. The factors considered included bufferpool size, page size, intermediate join indexes ordering criteria, bufferpool page replacement policies, among others. This analysis was based on real data taken from a GIS application for telecommunications, indexed on a R*-tree. Results of this work assess the way those factors affect spatial join performance and can be used for tuning such methods.

# Conteúdo

# Lista de Tabelas

# Lista de Figuras

# Capítulo 1

# Introdução

Temos assistido ao grande desenvolvimento da informática. Novas e diferentes aplicações surgem a cada dia, proporcionando maior produtividade às organizações e incrementando a economia, impulsionadas por máquinas cada vez mais velozes e softwares cada vez mais sofisticados. Neste contexto, destacamos as aplicações geográficas, onde a localização espacial é considerada ao lado de dados escalares (convencionais).

Aplicações nesta área são utilizadas em campos como agricultura de precisão, utilidade pública (telefonia, água e esgoto, distribuição de energia elétrica), controle ambiental, visão robótica, indexação e manipulação de imagens, cartografia, demografia, otimização de tráfego, dentre outras. Estas áreas são assistidas por Sistemas de Informações Geográficas (SIGs) [CCH+96], softwares que lidam com *dados espaciais.*

Sistemas de Informações Geográficas são sistemas utilizados para armazenar, manipular e analisar dados geográficos de forma automática. Estes dados representam objetos e fenômenos cuja característica fundamental à informação é a localização geográfica.

Dados espaciais possuem características que os tornam diferentes de dados convencionais. São necessários o desenvolvimento e o aperfeiçoamento de métodos que gerenciem e utilizem estes dados de forma eficiente. Dentre as operações mais importantes que se podem executar sobre dados espaciais, citamos as *junções espaciais.*

Embora haja muitos trabalhos que lidam com junções espaciais, poucos oferecem uma idéia dos fatores que influenciam o desempenho desta operação. Podemos ainda observar que, dentre as experiências feitas, poucas lidam com dados extraídos de uma aplicação real, somado ao fato de que as descrições dos testes são pouco precisas, ficando difícil uma quantificação dos resultados.

Neste contexto, este trabalho procura avaliar algumas propostas de junções espaciais com relação à influência de tamanhos de página, tamanhos de *bufferpool*, critérios de ordenação e outros fatores que possam afetar o desempenho destas operações. Em especial, procuramos investigar as propostas que tomam por base a R*-tree.

Este trabalho está organizado da seguinte forma: o capítulo 2 apresenta uma revisão dos critérios de classificação e métodos de indexação espacial, métodos de abstração de dados espaciais; o capítulo 3 apresenta os métodos de junção espacial propostos na literatura, bem como modelos analíticos de desempenho de junção espacial; o capítulo 4 apresenta a implementação dos testes, os resultados obtidos e discussões sobre os resultados; e finalmente, o capítulo 5 apresenta as conclusões e possíveis extensões a este trabalho.

# Capítulo 2

# Métodos de Acesso e Técnicas de Abstração de Objetos Espaciais

## 2.1 Introdução

Este capítulo apresenta uma visão geral das pesquisas realizadas na área de métodos de acesso espaciais. Não se pretende fazer uma compilação completa, tarefa extremamente difícil devido à quantidade de propostas, mas mostrar as técnicas e idéias utilizadas para o desenvolvimento destes métodos, bem como as principais características de cada família. A ênfase deste capítulo será nas propostas mais recentes, e em especial nos métodos mais apropriados a Sistemas de Informações Geográficas (SIGs), como os métodos baseados em árvores multiárias (seção 2.7). Portanto, métodos que lidam com altas dimensionalidades como a SR-tree [KS97a], a SS-tree [WJ96] e a X-tree [BKK96] não serão considerados. Esta revisão está embasada nos artigos originais dos autores e também em outras revisões, como o trabalho de [Car98] e [CM92].

O capítulo está organizado da seguinte forma: a seção 2.2 traz uma revisão das abstrações utilizadas para a representação de objetos espaciais no processo de indexação; a seção 2.3 discute algumas formas de se agrupar propostas de métodos de acesso espaciais; a seção 2.4 trata dos métodos que utilizam métodos de acesso convencionais no acesso a dados espaciais; a seção 2.5 apresenta alguns métodos que utilizam árvores binárias na indexação espacial; a seção 2.6 discute métodos baseados em estruturas *hash*; e, finalmente, a seção 2.7 discute métodos baseados em árvores multiárias, estruturas que são o foco de nosso trabalho.

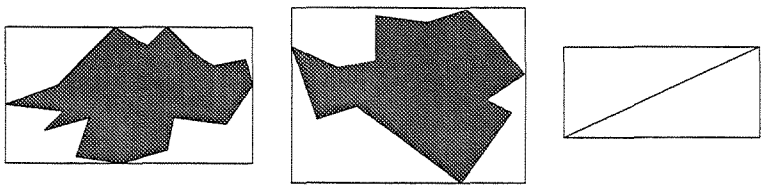


Figura 2.1: Alguns objetos espaciais e suas abstrações através de retângulos envolventes mínimos.

## 2.2 Abstração de objetos espaciais

Objetos indexados por métodos de acesso espaciais são freqüentemente representados como linhas e polígonos com grande quantidade de pontos. Essa geometria é excessivamente dispendiosa na indexação em termos de espaço de armazenamento e complexidade na avaliação de predicados. Estruturas específicas voltadas ao armazenamento de objetos espaciais na sua forma completa existem [MCD94, SK91] mas, embora no armazenamento todos os detalhes dos objetos sejam capturados, índices espaciais utilizam certas aproximações da geometria dos objetos, sem perder propriedades geométricas essenciais como a posição no espaço e a extensão em cada dimensão. As aproximações que garantem a manutenção dessas propriedades são chamadas de *conservativas* [BKSS94].

A abstração mais utilizada pelos métodos de acesso espaciais é o *MBR* (*minimum bounding rectangle*), ou *retângulo envolvente mínimo*. O MBR de um objeto em um espaço $k$-dimensional é o menor retângulo $k$-dimensional com lados paralelos aos eixos das dimensões que envolve completamente o objeto. A Figura 2.1 mostra alguns objetos e suas abstrações através de MBRs.

As duas maiores vantagens dos MBRs, que o tornaram amplamente utilizado, são o pequeno espaço utilizado em seu armazenamento (para um espaço bidimensional, são necessários dois pares de coordenadas) e a facilidade na avaliação de predicados espaciais. A estas vantagens deve-se somar o fato de ser uma abstração conservativa. Isto permite que relacionamentos entre MBRs possam ser deduzidos dos relacionamentos entre os objetos. Por exemplo, se dois objetos se interceptam, seus MBRs também se interceptam, o mesmo valendo para outros predicados como "contém" e "está contido". A recíproca, entratanto, não é verdadeira. Como exemplo, dados dois MBRs que se interceptam, não se pode afirmar com certeza que os objetos representados se interceptam, como ilustrado na Figura 2.2. Isto se deve ao fato de que pode haver espaço livre em cada MBR, ou seja, espaço não tomado pelo objeto representado pelo MBR, fato conhecido por *dead space*.

A diferença existente entre os objetos espaciais e suas abstrações pode gerar resultados diferentes na avaliação de predicados sobre o objeto e sua representação. Devido a este fator, consultas espaciais são executadas em duas fases:

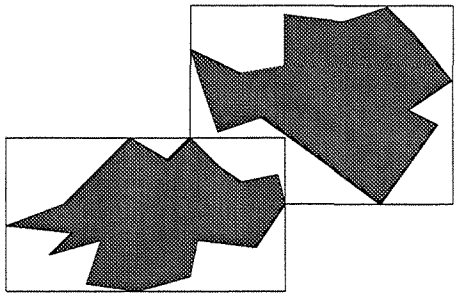


Figura 2.2: Dois MBRs que se interceptam, embora não haja interseção entre os objetos representados.

- filtragem, na qual um conjunto de objetos "candidatos" é recuperado, ou seja, aqueles cuja abstração satisfaz o predicado, embora o objeto em si possa não o satisfazer. Nesta fase, predicados específicos são substituídos por predicados mais gerais. Via de regra, o predicado utilizado nesta fase é a existência ou não de interseção. Por exemplo: se o predicado que está sendo avaliado é "toca", na filtragem ele é substituído por "intercepta", uma vez que, se dois objetos se tocam, é possível que suas abstrações se interceptem ao invés de se tocarem, e assim esses objetos seriam descartados;

- refinamento, na qual a geometria completa do objeto recuperado durante a filtragem é avaliada em relação ao predicado. Os objetos que satisfazem o predicado são devolvidos no conjunto resposta.

A recuperação da forma exata dos objetos selecionados na fase de filtragem e sua avaliação em relação ao predicado durante a fase de refinamento são muito dispendiosas. Além disso, muitos objetos selecionados na fase de filtragem serão descartados na fase de refinamento, ocorrência denominada *false hit*, devido ao *dead space* contido no MBR. Deste modo, é interessante diminuir o número de objetos candidatos que passarão para a fase de refinamento. Alguns autores [BKSS94, BK94] sugerem, além do MBR, a utilização de outras *aproximações conservativas*. Uma aproximação é dita conservativa se, e somente se, cada ponto dentro do contorno do objeto original está também na aproximação conservativa. As aproximações citadas pelos autores são as seguintes: o retângulo envolvente mínimo rotacionado *(rotated minimum bounding rectangle – RMBR)*, o casco convexo *(convex hull– CH)*, o polígono envolvente mínimo com $m$ vértices *(minimum bounding m-corner – m-C)*, o círculo envolvente mínimo *(minimum bounding circle – MBC)* e a elipse envolvente mínima *(minimum bounding elipse – MBE)*. A Figura 2.3 apresenta exemplos destas aproximações, apresentando entre colchetes o número de parâmetros necessários para armazená-las.

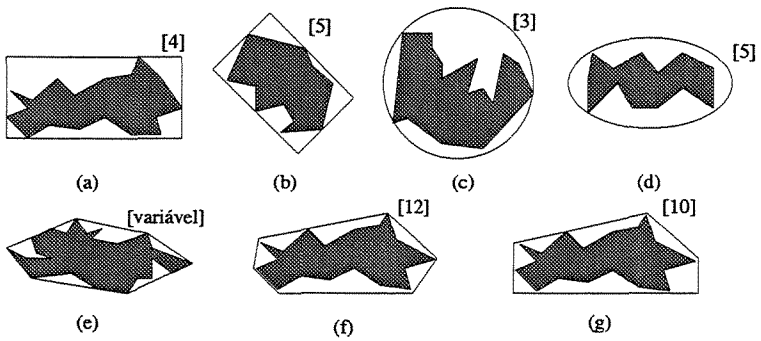



Figura 2.3: Algumas abstrações de objetos espaciais: (a) retângulo envolvente mínimo (MBR), (b) retângulo envolvente mínimo rotacionado, (c) círculo envolvente mínimo, (d) elipse envolvente mínima, (e) casco convexo, (f) polígono envolvente mínimo com 6 vértices, (g) polígono envolvente mínimo com 5 vértices.

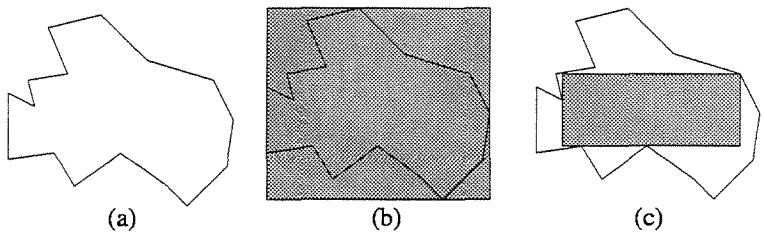



Figura 2.4: Um objeto (a) e suas aproximações conservativa (b) e progressiva (c).

Embora estas abstrações diminuam o *dead space* (exceto o MBC), seu uso acarreta maior espaço de armazenamento e maior complexidade na avaliação de consultas. Assim, a proposta dos autores é de armazenar juntamente com o MBR alguma destas abstrações, que seria utilizada em uma nova fase no processamento de consultas entre as fases de filtragem e refinamento, chamada de *aproximação*. Nesta etapa, os objetos recuperados durante a fase de filtragem através de seu MBR teriam sua outra abstração verificada em relação ao predicado avaliado e somente aqueles para os quais ele é satisfeito passariam para a fase de refinamento. Dentre as abstrações apresentadas pelos autores, o casco convexo é a que apresenta melhor desempenho em relação à acurácia, mas demanda grande espaço de armazenamento enquanto que a abstração que apresenta melhor relação (*false hits* identificados × espaço de armazenamento) é o polígono envolvente mínimo de 5 vértices.

Outro tipo de aproximação sugerida pelos autores é a *aproximação progressiva*. Um objeto poligonal é progressivamente aproximado se o conjunto de pontos da aproximação for um subconjunto do conjunto dos pontos do objeto. A Figura 2.4 ilustra os conceitos de aproximação conservativa e progressiva. Nota-se que, se duas aproximações progressivas se interceptam, então os objetos por elas representados também se interceptam. As

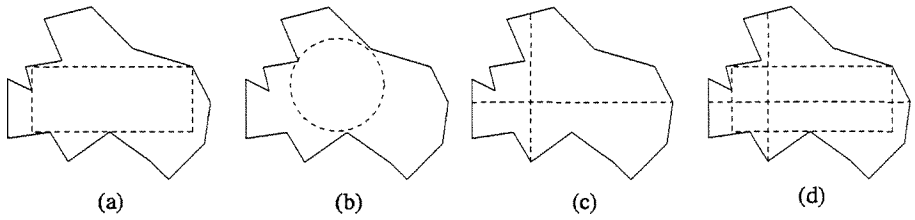



Figura 2.5: Aproximações progressivas: (a) retângulo inscrito máximo, (b) o círculo inscrito máximo, (c) segmentos de linha inscritos máximos, (d) combinação de MER e EL.

aproximações progressivas investigadas pelos autores foram: o círculo inscrito máximo (*maximum enclosed circle - MEC*), o retângulo inscrito máximo (*maximum enclosed rectangle - MER*), os segmentos de linha inscritos máximos (*maximum enclosed line segments - EL*) e uma combinação de MER e EL. A Figura 2.5 mostra alguns exemplos de aproximações progressivas.

É bom notar que o cálculo de aproximações progressivas é mais caro que das aproximações conservativas. Os autores propõem a utilização de aproximações conservativas, em especial a 5-C (veja Figura 2.3 na eliminação de *false hits* no conjunto gerado na fase de filtragem. Após a eliminação dos *false hits* os autores sugerem as aproximações progressivas, como o MEC ou MER, para a detecção de *hits*, ao invés de recuperar a forma real do objeto para avaliá-lo en relação ao predicado. Apenas os objetos que não fossem detectados na utilização de aproximação progressiva teriam sua geometria recuperada e verificada quanto ao predicado.

Outro trabalho que busca melhorar o desempenho de consultas espaciais através da diminuição do *dead space* é apresentado em [LLRC96], no qual os autores propõem a representação de um objeto através *retângulos envolventes mínimos decompostos* (DMBR). Nesta técnica, o polígono a ser representado é dividido em dois subpolígonos correspondentes às regiões direita e esquerda de seu MBR, gerando então dois MBRs novos, chamados DMBRs. Esta operação é repetida recursivamente, alterando-se o eixo de partição, até que uma dada restrição controlada por um parâmetro $g$ seja atingido. Uma nova divisão só é permitida se a área do DMBR a ser dividido for maior do que $2^{-g}$ da área do MBR original. O *dead space* diminui à medida que $g$ cresce, mas o número de DMBRs também cresce, o que pode trazer a uma deterioração do desempenho e aumento do espaço de armazenamento. Os autores relatam que os melhores desempenhos foram obtidos para valores de $g$ entre 3 e 4. A Figura 2.6 ilustra um processo de decomposição para $g = 2$, ou seja, um DMBR só é particionado novamente se sua área for maior do que 25% da área do MBR inicial.

Outras duas técnicas utilizadas para decomposição de objetos espaciais são as seguintes:

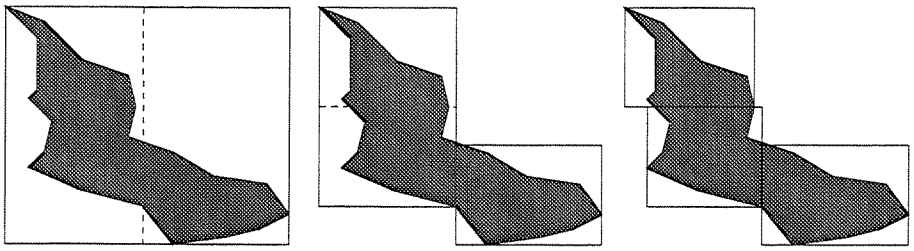


Figura 2.6: Decomposição de um objeto em DMBRs, com $g = 2$. O eixo de partição é representado através de linhas tracejadas.

- decomposição regular, na qual o espaço onde os objetos estão dispostos é divido em uma grade regular. O objeto é então representado pelo conjunto de células que ele intercepta;
- decomposição estrutural, que utiliza o contorno do objeto poligonal para orientar a decomposição. Um exemplo é a utilização dos vértices de um objeto para particioná-lo em trapezóides, triângulos ou polígonos convexos

Estas técnicas, entretanto, possuem a inconveniência de gerar grande número de componentes, o que demanda mais espaço de armazenamento.

Em [SLS95], Stefanakis, Lee e Sellis propõem uma outra técnica para a extensão de métodos de acesso a pontos, chamada de *abstração*. Esta técnica representa os MBRs dos objetos espaciais através de pontos no mesmo espaço onde os MBRs estão dispostos. Dois modos de representação são sugeridos:

- *por vértices*, no qual o MBR é substituído pelas coordenadas de seus quatro cantos;
- *central*, no qual é utilizado o ponto central da figura.

Esta técnica preserva a localização dos objetos mas não preserva sua extensão, causando erros durante a fase de filtragem. Para resolver este problema, os autores sugerem uma *janela de consulta estendida*, cuja idéia é aumentar o tamanho da janela de consulta original, se necessário, de forma que todos os objetos que cruzam a janela original sejam recuperados na filtragem. A janela de consulta estendida $q$ é calculada através dos seguintes dados:

- $dX_{max}$, que é o comprimento do maior lado na dimensão $X$ entre os MBRs armazenados;
- $dY_{max}$, que é o comprimento do maior lado na dimensão $Y$ entre os MBRs armazenados;

- $dX_q$, que é o comprimento da janela de consulta em relação ao eixo $X$;
- $dY_q$, que é o comprimento da janela de consulta em relação ao eixo $Y$;

A correção das consultas é garantida se $dX_q \geq dX_{max}$ e $dY_q \geq dY_{max}$. Este aumento só será necessário se um dos lados da janela de consulta for menor que o maior lado de MBR na extensão correspondente. Se for utilizada a representação central, o aumento da janela de consulta é obrigatório. Assim, a nova janela de consulta, com ponto central igual ao da janela de consulta anterior, terá a extensão $dX_q + dX_{max}$ no eixo $X$ e $dY_q + dY_{max}$ no eixo $Y$.

Embora tenham sido apresentadas várias técnicas alternativas para a representação de objetos espaciais, o MBR continua sendo o mais utilizado devido à simplicidade com que predicados podem ser avaliados, pouco espaço de armazenamento exigido e facilidade de se determiná-lo.

## 2.3 Classificação de métodos de acesso espaciais

Existem duas propostas de classificação para os métodos de acesso espaciais: a primeira [SRF87, MCD94, Ooi90] baseia-se na dimensão dos dados indexados, enquanto que a segunda baseia-se nos métodos originais sobre os quais se baseiam os índices espaciais [CM92].

### 2.3.1 Classificação baseada na dimensão dos dados

Esta classificação divide os métodos de acesso espaciais em *métodos de acesso a pontos* e *métodos de acesso a objetos de dimensão não zero*. Alguns trabalhos [Ooi90, KSSS89] propõem que apenas os métodos do segundo grupo sejam considerados métodos de acessos espacias, referindo-se aos do primeiro apenas como métodos de acesso a pontos. Estes grupos, como veremos a seguir, sofrem subdivisões.

#### Métodos de acesso a pontos

Métodos de acesso a pontos indexam dados através da divisão do espaço em sub-regiões disjuntas de forma que cada sub-região tenha, no máximo, $n$ pontos. Cada sub-região é então associada a uma página de disco capaz de armazenar $n$ pontos. Uma variação desta abordagem é a divisão do espaço de acordo com o posicionamento dos pontos no espaço. Neste caso, cada nó da estrutura corresponde a apenas um ponto.

A inserção de um novo ponto pode fazer com que a capacidade de armazenamento de um nó seja ultrapassada. Este fato causa o particionamento de uma região, evento

conhecido por *split*. O *split* ocorre todas as vezes que se insere um ponto nos métodos que subdividem o espaço de acordo com a localização dos pontos. O particionamento da região é feito através da introdução de um ou mais hiperplanos $k-1$-dimensionais no espaço $k$-dimensional onde se encontram os pontos indexados, resultando em duas novas sub-regiões disjuntas.

Sellis, Roussopoulos e Faloutsos [SRF87] apresentam três classificações para os métodos de acesso a pontos, de acordo com as características dos algoritmos de *split* utilizados:

1. Baseada na posição do(s) hiperplanos(s)
   - métodos fixos: a posição do hiperplano é predeterminada.
   - métodos adaptáveis: a posição do hiperplano é determinada pela posição do ponto inserido.
2. Baseada no número de dimensões em que são inseridos os hiperplanos
   - métodos que particionam o espaço em apenas uma dimensão: neste método há um rodízio no eixo de partição, ou seja, se a região a ser divida foi originada de uma partição na dimensão $d$, então, ela é particionada na dimensão $d+1$. Se $d$ for igual ao número de dimensões, então a região é particionada em relação à primeira dimensão.
   - métodos que particionam o espaço em $k$ dimensões: neste método, o espaço $k$-dimensional é particionado em todas as $k$ dimensões.
3. Baseada na localidade da partição
   - métodos de grade: o hiperplano divide não apenas a região de partição, mas também todas as regiões em sua direção
   - métodos de divisão hierárquica: o hiperplano divide apenas a região que está sofrendo o *split*, levando a uma decomposição hierárquica do espaço.

Em [KSSS89], os autores propõem uma classificação baseada nos seguintes fatores em relação à partição do espaço:

- se as regiões são disjuntas;
- se as regiões são retangulares;
- se a união das regiões corresponde a todo o espaço onde estão dispostos os objetos.

A partir destes fatores, os autores propõem uma classificação em classes de acordo com a Tabela 2.1:

| | Propriedades | | |
|---|---|---|---|
| Classe | Regiões retangulares | Cobertura completa | Regiões disjuntas |
| $C_1$ | sim | sim | sim |
| $C_2$ | sim | sim | não |
| $C_3$ | sim | não | sim |
| $C_4$ | não | sim | sim |

Tabela 2.1: Classificação de métodos de acesso espaciais de acordo com as caraterísticas das regiões nas quais o espaço é dividido.

### Métodos de acesso a objetos de dimensão não zero

Em geral, os métodos de acesso a objetos de dimensão não zero são divididos em subclasses de acordo com a técnica original que foi utilizada para derivá-los. Os nomes destas subclasses variam de trabalho para trabalho, bem como a forma de agrupá-las. Ooi, Sacks-Davis e Han [OSH93], agrupam estes métodos da seguinte forma:

1. Transformação

   - transformação para um espaço de maior dimensão: nesta técnica, objetos com $n$ vértices em um espaço $k$-dimensional são mapeados para um espaço $nk$-dimensional. Por exemplo, um MBR representado pelas coordenadas de seu vértice inferior esquerdo $(x_1, y_1)$ e $(x_2, y_2)$ é representado como um ponto em um espaço quadridimensional pelas coordenadas $(x_1, y_1, x_2, y_2)$. Depois de transformados, os pontos podem ser armazenados em métodos de acesso a pontos. Embora simples, esta técnica possui a inconveniência de não preservar distâncias entre os objetos indexados, ou seja, objetos próximos em um espaço $k$-dimensional podem ficar distantes em um espaço $nk$-dimensional e vice-versa;
   - transformação para o espaço unidimensional: nesta técnica, o espaço é dividido em uma grade de células do mesmo tamanho. As células, por sua vez, são numeradas através de alguma *space-filling curve* (veja seção 2.4.1) e ordenadas. Cada objeto é então representado pelo conjunto dos números das células que intercepta. Estes números são tratados como coordenadas de pontos em um espaço unidimensional e indexados através de algum método de acesso convencional. Uma desvantagem deste método é que um objeto pode ser representado por vários conjuntos disjuntos de pontos, fazendo com que tenha que ser referenciado várias vezes.

2. Divisão do espaço nativo sem sobreposição

Nesta abordagem, um espaço $k$-dimensional é particionado em sub-regiões disjuntas e a abstração do objeto é representada em todas as sub-regiões que ela intercepta. A representação pode ser feita de duas formas:

- duplicação do objeto, onde o identificador do objeto é duplicado e armazenado em todas as sub-regiões que intercepta;
- *object clipping*, onde o objeto é decomposto em vários objetos menores e disjuntos, de forma que cada sub-objeto esteja completamente contido em uma única sub-região.

A propriedade mais importante da duplicação de objetos e do *object clipping* é que as estruturas de dados utilizadas são extensões diretas de estruturas de indexação de pontos. Além disso, pontos e objetos de tamanho não zero podem ser armazenados juntos sem ter que modificar a estrutura. A desvantagem é a possível repetição de objetos, que requer mais espaço de armazenamento e maior complexidade nas rotinas de inserção e remoção. Outra limitação é que a densidade[1] do espaço deve ser menor que a capacidade de armazenamento da página. Como exemplo, considere o espaço $S$ representado na Figura 2.7 (a), cuja densidade máxima suportada é de quatro objetos. A inserção do retângulo $r$ faz com que não seja possível a subdivisão do espaço, uma vez que um particionamento que respeite a capacidade máxima não é possível devido à área de interseção sombreada, mostrada na Figura 2.7 (b).

3. Divisão do espaço nativo com sobreposição

   Da mesma forma que a anterior, esta técnica divide o espaço em sub-regiões e as organiza em um índice hierárquico, mas permite que elas se sobreponham. A vantagem desta técnica em relação à primeira é que cada objeto é representado uma única vez no índice. A desvantagem está no fato de que uma consulta realizada na área de interseção entre duas ou mais sub-regiões levará a uma busca em todos os caminhos correpondentes a estas sub-regiões no índice hierárquico.

Segundo [OSH93], alguns índices usam mais de uma técnica para estender os métodos de acesso a pontos para a indexação de objetos de maior dimensão de forma que, uma técnica compense os pontos fracos de outra. Entretanto, isto pode ter o efeito contrário, fazendo com que o método desenvolvido herde os pontos fracos de ambas

[1]Entende-se por *densidade* o número máximo de objetos que se interceptam em determinado ponto

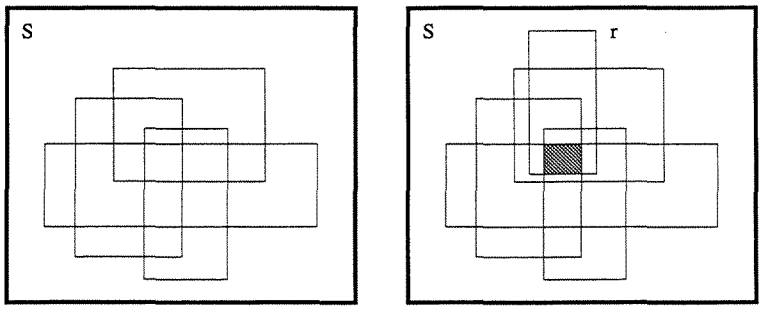



Figura 2.7: Um espaço cuja densidade máxima suportada é quatro (a) e a inserção de um novo MBR $r$, que causa um aumento da densidade, representado pela área mais escura (b).

### 2.3.2 Classificação baseada nos métodos de acesso convencionais de origem

Esta classificação foi proposta por Cox em [CM92] e por Ooi, Sacks-Davis e Han em [OSH93]. Cox diz que tanto os métodos utilizados nos métodos de acesso a pontos quanto os métodos de acesso a dados de dimensão não zero são extensões de estruturas de dados utilizadas no armazenamento de dados escalares. As justificativas para esta proposta de classificação, segundo Cox, são as seguintes:

- a classificação baseada na dimensão dos dados não existe, uma vez que é possível indexar objetos de dimensão não zero em estruturas de acesso a pontos através da técnica de transformação. Da mesma forma, pontos podem ser tratados como MBRs de extensão nula e indexados em uma estrutura de acesso a objetos de dimensão não zero;

- muitas características das estruturas convencionais são conservadas nas estruturas estendidas, permitindo maior facilidade na determinação de quais métodos são mais apropriados através de relações de flexibilidade, desempenho e complexidade dos métodos base.

Cox classifica os métodos de acesso espaciais em três categorias: derivados de árvores binárias, derivados de estruturas *hash* e derivados de árvores multiárias. Ooi, Sacks-Davis e Han chamam estes últimos de "derivados de árvores $B^+$", além de definir um quarto grupo englobando os que utilizam *space-filling curves*. Carneiro [Car98] sugere que este último grupo seja denominado de *métodos que utilizam índices unidimensionais (convencionais)*, devido à existência de métodos como em a Hilbert R-tree [KF94] que utilizam

*space-filling curves* apenas para ordenar os MBRs dos objetos, não transformando-os em pontos.

Neste trabalho, utilizaremos a classificação proposta por Cox acrescida do grupo sugerido por Carneiro e Ooi, Sacks-Davis e Han. Daremos uma certa ênfase aos métodos derivados de árvores multiárias, uma vez que o enfoque de nosso trabalho se concentra nestes métodos.

## 2.4 Métodos que utilizam índices convencionais

Métodos convencionais podem ser utilizados para armazenar dados espaciais se, para isso, os dados sofrerem uma transformação para o espaço unidimensional ou algum tipo de ordenação linear. A vantagem do uso de métodos convencionais está no fato de que todos os SGBDs comerciais suportam ao menos um destes índices, o que não ocorre com métodos de acesso espaciais.

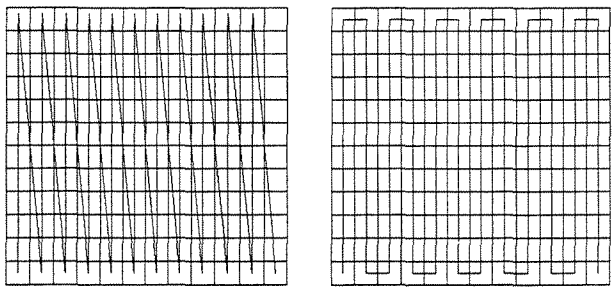
Nas próximas subseções, descrevemos alguns métodos de acesso espaciais que utilizam métodos de acessos convencionais na indexação dos dados.

### 2.4.1 Transformação através de *space-filling curves*

Informalmente, uma *space-filling curve* é um caminho contínuo que visita todos as células de uma grade $k$-dimensional exatamente uma vez sem nunca se cruzar. Elas provêem um método de se ordenar linearmente as células de uma grade, preservando na medida do possível a distância entre elas, ou seja, células próximas no espaço $k$-dimensional são armazenadas próximas na ordenação linear.

A ordenação gerada por uma *space-filling curve* será tanto melhor quanto mais eficaz ela for na preservação das distâncias. Isto se deve ao fato de que pontos que estão próximos no espaço $k$-dimensional têm maior probabilidade de serem recuperados juntos em uma consulta do que aqueles que estão distantes entre si. Se dois pontos próximos no espaço são armazenados próximos em uma *space-filling curve*, a probabilidade de que venham a ser armazenados em uma mesma página ou em páginas contíguas é muito grande, o que diminui o tempo de acesso a disco. Em [FR89] Faloutsos e Roseman propõem o uso de *space-filling curves* e em especial a curva de Hilbert para o projeto de mapeamentos que preservam a distância.

A seguir apresentamos algumas *space-filling curves*. Por simplicidade serão considerados apenas espaços bidimensionais, embora possam ser utilizadas em espaços de qualquer dimensão.



(a) (b)

Figura 2.8: Percurso por colunas (a) e percurso por colunas com sentido alternado (b).

### Percurso por colunas

O percurso por colunas (ou por linhas) é a forma mais simples de se ordenar as células de uma grade. A coordenada linear, calculada através da expressão $x \times nc_y + y$, é definida por sua posição seqüencial na linha de percurso. Um exemplo deste tipo de percurso é mostrado na Figura 2.8.

Outra forma de se percorrer as colunas de uma grade é mostrada na Figura 2.8 (b), onde o percurso é invertido a cada mudança de coluna. Esta curva preserva melhor as distâncias do que a anterior, uma vez que dois pontos consecutivos na ordenação linear também são consecutivos no espaço. A coordenada linear de uma célula bidimensional é dada por $x \times nc_y + y$, para valores pares de $x$ e $(x+1) \times nc_y - y - 1$ para valores ímpares de $x$.

### Curva de Peano ou Curva Z

A curva $Z$ foi proposta para o mapeamento de células multidimensionais em [OM84]. A coordenada unidimensional é calculada através da intercalação dos bits das representações binárias de suas coordenadas em cada dimensão, definida em uma função de embaralhamento cuja única restrição é manter a ordem entre os bits de cada coordenada. Possíveis funções de embaralhamento, dadas duas cadeias de bits $X$ e $Y$, são:

1. tomar alternadamente um bit de cada cadeia, iniciando pela cadeia $X$;
2. tomar alternadamente um bit de cada cadeia, iniciando pela cadeia $Y$;
3. concatenar os bits da cadeia $Y$ à esquerda dos bits da cadeia $X$;

Embora existam várias funções, a mais comum é a primeira. A curva correspondente a esta função em uma grade $2 \times 2$ é denominada *curva de ordem 1.* A partir dela, pode-se

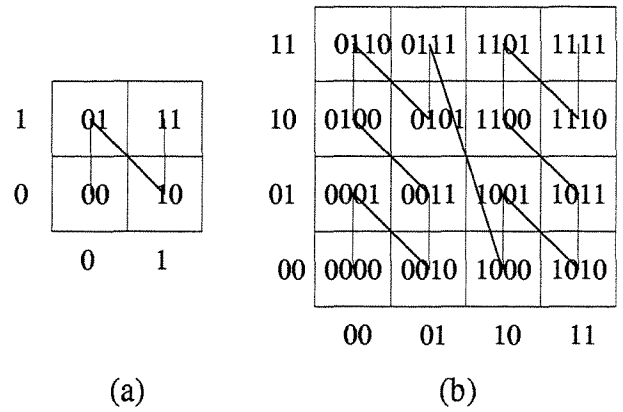



Figura 2.9: Caminho feito por uma curva $Z$ em uma grade $2 \times 2$ (a) e uma grade $4 \times 4$ (b).

gerar qualquer curva de ordem $i$, que percorre uma grade $2^i \times 2^i$. Na Figura 2.9 mostramos o resultado desta função em grades $2 \times 2$ (ordem 1) e $4 \times 4$ (ordem 2).

### Curva de Código Binário Gray Refletido

Em um código binário Gray (*Gray code*), as cadeias de bits são seqüenciadas de forma que duas cadeias consecutivas tenham exatamente um bit divergente. A forma mais utilizada de se organizar as cadeias desta forma é através dos *códigos binários Gray refletidos* (RBG – *reflectec binary Gray code*).

Um código RBG ($G(n)$) com cadeias de $n$ bits pode ser denotado por uma matriz binária de $n \times 2^n$, onde cada linha $G_{n,i}$ é uma cadeia de bits do código.

Assim, para cadeias de 1 bit, a matriz que define o código Gray refletido é:

$$G = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

Se a matriz que representa um código Gray com $n$ bits é representada por:

$$G(n) = \begin{bmatrix} G_{n,0} \\ G_{n,1} \\ \vdots \\ G_{n,2^n-1} \end{bmatrix}$$

então a matriz de código Gray refletido para cadeias de $n+1$ bits é obtida fazendo:

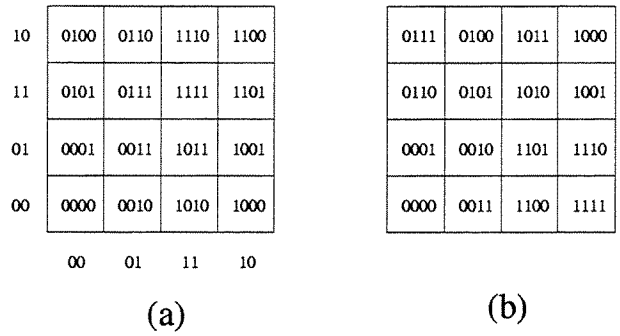



Figura 2.10: Códigos Gray refletidos para uma grade $4 \times 4$ gerados através da intercalação dos bits das coordenadas de cada célula. Note que tanto as coordenadas quanto os códigos das células diferem em exatamente um bit (a), e seus equivalentes binários (b).

$$G(n) = \begin{bmatrix} 0G_{n,0} \\ 0G_{n,1} \\ \vdots \\ 0G_{n,2^n-1} \\ 1G_{n,2^n-1} \\ \vdots \\ 1G_{n,1} \\ 1G_{n,0} \end{bmatrix}.$$

Dada a matriz geral, a curva de código Gray refletido é calculada enumerando-se as linhas e colunas da grade utilizando o código binário Gray refletido. O código de cada célula é derivado através da intercalação dos bits de cada coordenada (a Figura 2.10 mostra esta intercalação para uma grade $4 \times 4$). O cálculo da posição de uma célula na curva RBG é feito através da transformação de seu código para seu equivalente decimal (ou binário).

Faloutsos, em [Fal86], propõe o uso da curva RBG em funções de *hashing* e na ordenação linear de células multidimensionais, em substituição à curva $Z$. Na Figura 2.11 apresentamos as curvas RGB de ordem 1, ordem 2 e ordem 3, respectivamente.

### Curva de Hilbert

A desvantagem das curvas anteriores está no fato de que nem sempre dois elementos consecutivos na curva encontram-se próximos no espaço multidimensional, devido aos

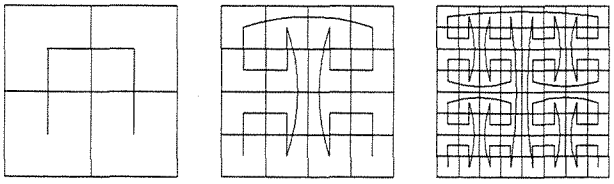


Figura 2.11: Curvas RBG de ordem 1, ordem 2 e ordem 3.

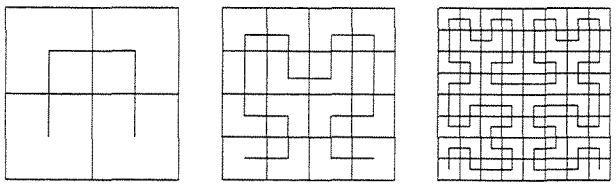


Figura 2.12: Curvas de Hilbert de ordem 1, ordem 2 e ordem 3, respectivamente.

"saltos" entre colunas ou linhas. Embora o percurso por colunas com sentidos alternados não apresente este inconveniente, duas células da grade multimensional podem ter seus correspondentes na curva muito distantes entre si.

Desta forma, Faloutsos e Roseman [FR89] e Jagadish [Jag90] propõem a utilização da curva de Hilbert em substituição às outras curvas na ordenação de células multidimensionais. Estes trabalhos mostram que a curva de Hilbert é a que oferece melhor desempenho no agrupamento de células próximas no espaço em relação ao espaço unidimensional.

O cálculo da coordenada linear de uma célula é feito através da intercalação dos bits das coordenadas em cada dimensão e algumas transformações feitas para refletir as rotações das coordenadas. A Figura 2.12 apresenta as curvas de Hilbert de ordem 1, 2 e 3, respectivamente.

### Utilização de *space-filling curves* na indexação de objetos de dimensão não zero

Depois de transformados para o espaço unidimensional, a indexação de pontos em algum método de acesso convencional é direta. O mesmo não se aplica a objetos de dimensões maiores do que zero, uma vez que um objeto pode interceptar mais de uma célula da grade.

Assim, para se representar um polígono, utiliza-se a técnica de decomposição regular (seção 2.2), onde o tamanho da célula é definido de acordo com a precisão desejada. O polígono é representado pelas coordenadas das células que intercepta. Isto, entretanto, pode acarretar que um objeto possa ser representado por um grande número de células.

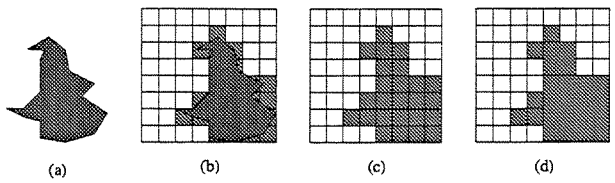



Figura 2.13: Processo de representação de um polígono através de coordenadas lineares de uma curva $Z$.

Para diminuir este problema, as células são agrupadas, na medida do possível, em conjuntos de supercélulas.

Como exemplo, consideremos uma grade formada pelo particionamento regular do espaço, de forma que tenhamos $2^n \times 2^n$ células, sendo que a coordenada linear de cada célula é uma cadeia de $n$ bits. Uma supercélula é definida como um conjunto maximal $S_i$ do conjunto de células $P$ que representa um objeto, tal que:

- $|S_i| = 2^k$, $1 \leq k_i \leq 2n$;
- os $2n - k_i$ bits mais significativos são comuns às coordenadas lineares de todos os elementos de $S_i$.

Desta forma, um polígono passa a ser representado pelas coordenadas lineares das células que não participam de nenhuma supercélula, mais as coordenadas lineares das supercélulas que o compõem. É importante ressaltar que para se fazer comparações entre coordenadas, são considerados apenas os bits mais significativos das cadeias maiores, por exemplo: 110 é maior que 1011, pois 110 é maior do que 101.

A Figura 2.13 ilustra este processo: em (a) temos o objeto de dimensão não zero; em (b), apresentamos a sobreposição deste objeto em uma grade regular; em (c) temos todas as células interceptadas pelo objeto e finalmente em (d) temos o conjunto de células e supercélulas que representam o objeto. A Tabela 2.2 mostra o conjunto de coordenadas que representam o objeto e o agrupamento de células que o originou em relação às coordenadas $Z$ das células.

As operações de inclusão ou exclusão de um objeto são feitas através do cálculo das coordenadas dos pontos correspondentes no espaço unidimensional seguida da inclusão ou exclusão destas no método de acesso convencional.

Consultas para verificar a existência ou não de objetos são mais simples: basta recuperar a coordenada linear de qualquer ponto contido em uma das células que representam o objeto. Esta coordenada será uma cadeia de bits que estará contida na coordenada de alguma célula ou supercélula que representa o objeto. Entretanto, como mais de um objeto

| Coordenada | Agrupamento |
|---|---|
| 001001 | Célula 001001 |
| 001011 | Célula 001011 |
| 001110 | Célula 001110 |
| 011011 | Célula 011011 |
| 10 | Células 100000 a 101111 |
| 1100 | Células 110000 a 110010 |
| 110100 | Célula 110100 |

Tabela 2.2: Coordenadas lineares $Z$ que representam o objeto da Figura 2.13 através das células e supercélulas interceptadas.

pode estar na mesma célula ou supercélula, é necessária uma verificação de identificadores para se chegar a uma resposta correta.

### 2.4.2 DOT

O DOT (*Double Transformation*), proposto por Faloutsos e Rong [FR91], foi desenvolvido com o intuito de indexar objetos espaciais utilizando índices convencionais sem a geração das várias entradas para um mesmo objeto, como o que ocorre com os métodos que utilizam transformação através de *space-filling curves*.

Os objetos a serem indexados sofrem duas transformações: na primeira, seu MBR no espaço $k$-dimensional é transformado em um ponto de um espaço $2k$-dimensional. Na segunda, este ponto é transformado em outro em um espaço unidimensional através de uma *space-filling curve*. Esta coordenada é então indexada por um método de acesso convencional.

*Range queries* são executadas transformando-se as janelas de consulta para o espaço $2k$-dimensional, para depois se tornarem um conjunto de intervalos no espaço unidimensional. Como exemplo, considere o espaço unidimensional $S$ (consideraremos espaço unidimensional para maior simplicidade) da Figura 2.14 (a), cujos limites são $S_{min}$ e $S_{max}$, onde estão dispostos os objetos $A$, $B$ e a janela de consulta $Q$. A Figura 2.14 (b) mostra a primeira transformação dos objetos do espaço $S$ para o espaço bidimensional $S'$, cujas coordenadas mínimas e máximas são $S_{min}$ e $S_{max}$ respectivamente, em cada eixo. No espaço bidimensional, o eixo $X$ (horizontal) representa as coordenadas mínimas dos objetos, enquanto que o eixo $Y$ (vertical) representa as máximas.

A janela de consulta no espaço bidimensional apresentado na Figura 2.14 é limitada pelas seguintes curvas:

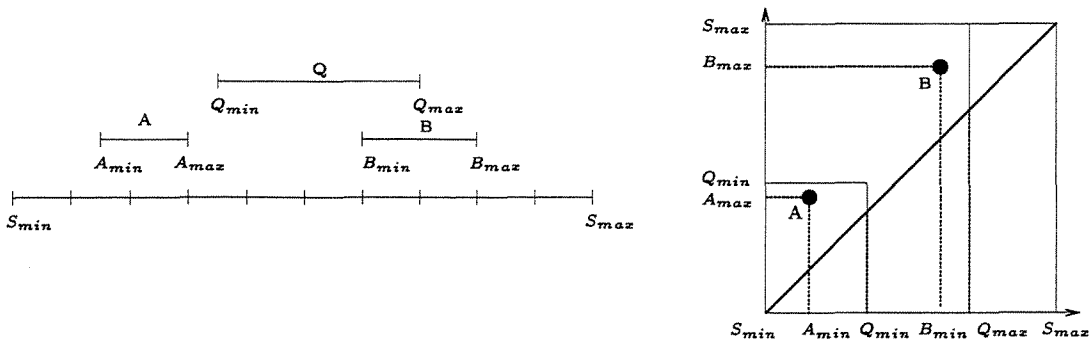



Figura 2.14: Um nó de R$^+$-tree com seus filhos (a) e os novos nós com o eixo de partição $L$ (b).

- $X = S_{min}$, pois não há segmento de reta que comece antes do início do espaço considerado;
- $X = Q_{max}$, pois um segmento de reta deve iniciar no máximo em $Q_{max}$ para que intercepte a janela de consulta;
- $Y = Q_{min}$, pois um segmento de reta deve terminar no mínimo em $Q_{min}$ para que intercepte a janela de consulta;
- $Y = S_{max}$, pois não há segmento de reta que termine após o final do espaço considerado;
- $X = Y$ pois, para qualquer segmento de reta, $X \leq Y$.

### 2.4.3 2dMAP21

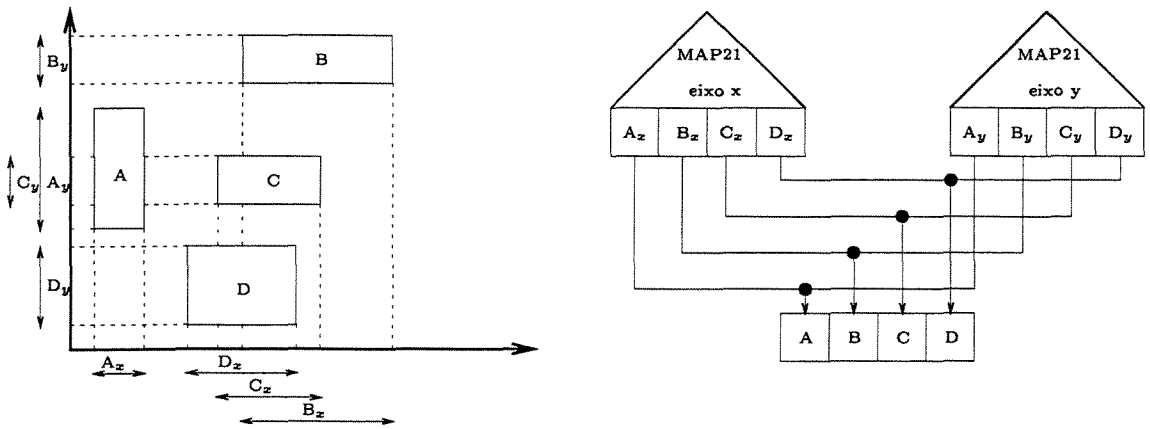
A 2dMAP21, de Nascimento e Dunham [ND97], é uma estrutura baseada na MAP21 [NDK96], uma estrutura criada para a indexação de intervalos de tempo representados como segmentos de reta.

A MAP21 utiliza uma função $F$ para transformar os pontos inicial e final $I_i^k$ e $I_f^k$ de cada intervalo de tempo $I^k$ em um único ponto, que depois é armazenado em uma B$^+$-tree. A função $F(I^k)$, que mapeia o intervalo $I^K$, é definida por

$$F(I^k) = F(I_i^k, I_f^k) = I_i^k \times 10^\alpha + I_f^k$$

onde $\alpha$ é o número máximo de dígitos necessários para representar a coordenada final de qualquer intervalo.

A função $F$ possui as seguintes propriedades:




(a) (b)

Figura 2.15: Objetos de um espaço decompostos em seus eixos $X$ e $Y$ (a) e sua representação em uma 2dMAP21 (b).

- mapeia intervalos distintos em pontos distintos;

- cria uma ordenação entre os intervalos, ou seja, dados dois intervalos $I^j = [I_i^j, I_f^j]$ e $I^k = [I_i^k, I_f^k]$, se $I_i^j < I_i^k$ então $F(I^j) < F(I^k)$; e se $I_i^j = I_i^k$ e $I_f^j < I_f^k$ então $F(I^j) < F(I^k)$

Para a indexação de retângulos, a 2dMAP21 utiliza uma MAP21 para cada dimensão. Retângulos são decompostos em suas projeções nos eixos horizontal e vertical (veja a Figura 2.15) e os intervalos obtidos através destas projeções são transformados em pontos e armazenados na árvore $B^+$ correspondente.

Consultas para determinação de interseção são feitas da seguinte forma: a janela de consulta é decomposta nas suas projeções vertical e horizontal. Cada projeção é utilizada para realizar uma consulta de interseção na estrutura MAP21 correspondente, gerando um conjunto de registros relativos a cada eixo. O conjunto resposta é dado pela interseção dos registros de cada conjunto anteriormente gerado. Consultas para identificação de retângulos contidos na janela de consultas são feitas da mesma maneira.

Além de herdar todo o suporte exitente para a $B^+$-tree, a 2dMAP21 é facilmente paralelizável, uma vez que as MAP21 que a compõem podem ser armazenadas em discos diferentes, fazendo com que uma consulta possa ser feita paralelamente em ambas.

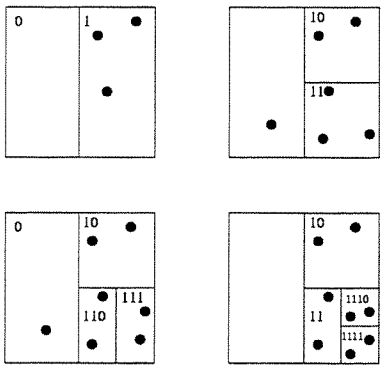



Figura 2.16: O particionamento do espaço representado por uma G-tree, com número máximo de entradas $M = 3$.

### 2.4.4 G-tree

A G-tree [Kum94] é uma estrutura criada para a indexação de pontos. Este método particiona o espaço hierarquicamente em regiões disjuntas, sendo que cada partição corresponde a uma página de disco ou *bucket*, comportando um número máximo de entradas $M$.

Caso a inserção de um novo ponto cause *overflow* na região na qual foi inserido, esta região é dividida em duas novas sub-regiões de mesmo tamanho. Esta divisão é feita em apenas uma das dimensões do espaço $k$-dimensional. O eixo é escolhido ciclicamente, ou seja, se uma região foi gerada por uma partição no eixo $i$, ela será particionada na dimensão $i + 1$.

A Figura 2.16 ilustra o processo de particionamento de um espaço bidimensional, para $M = 3$. Inicialmente, o espaço é dividido em duas regiões, cujos identificadores são 0 e 1. Cada vez que uma região é subdividida, as duas novas regiões são identificadas herdando os bits da região "pai" e concatenando-se 0 ou 1, para identificar a primeira região ou a segunda, respectivamente.

Os identificadores das regiões são strings de bits (note a semelhança com o particionamento com a curva $Z$), o que permite que as partições sejam ordenadas linearmente e armazenadas em uma árvore $B^+$. Para economizar espaço, as partições que não contêm pontos não são armazenadas.

A inserção de um ponto é feita identificando-se inicialmente se a partição a que pertence o mesmo já existe na árvore $B^+$. Uma vez que não se sabe antecipadamente que partição é esta, calcula-se uma partição inicial $P_apr$ e assume-se que o ponto será inserido numa partição de tamanho igual à menor partição criada até o momento e com o mesmo número de bits no identificador. A seguir, procura-se na árvore $B^+$ uma partição tal que $P_apr \subseteq P$.

Se a partição foi encontrada e ainda não atingiu sua capacidade máxima, o ponto é inserido. Caso seja necessário um *split*, a partição é dividida, excluída da árvore, e os pontos que continha são redistribuídos. Cada subpartição é então reinserida na árvore, desde que a subpartição não esteja vazia. Um novo valor de $P_apr$ é associado ao ponto a ser inserido e todo o processo se repete até a inserção do ponto.

Caso a partição procurada não exista, é necessário criá-la. A nova partição é a própria $P_apr$ ou sua ancestral de maior área, que não intercepte partições já existentes na estrutura.

A remoção de um ponto é feita da seguinte forma: inicialmente, identifica-se a partição $P$ a que pertence o ponto, procedimento semelhante ao efetuado na inclusão. Depois da remoção de um ponto, pode ser necessária uma junção de partições, caso o número de pontos em $P$ mais o número de pontos em $comp(P)$[2] for menor ou igual à capacidade máxima de uma página. Neste caso, a partição pai de $P$ é inserida da árvore $B^+$ e os pontos restantes de $P$, se existirem, e de $comp(P)$ são associados a ela. $P$ é excluída da estrutura, assim como $comp(P)$. Como o complemento do pai de $P$ pode não estar na árvore, estas junções de partições podem continuar recursivamente.

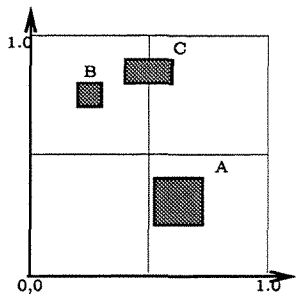
A consulta para verificação da existência de um ponto se resume na identificação da partição a que pertence e a uma busca na página a que pertence. *Range queries* são executadas da seguinte forma: inicialmente identificam-se as partições com menor e maior identificador que podem interceptar a janela de consulta. Em seguida, pesquisa-se a G-tree: cada partição contida no intervalo é testada para se determinar se ela está contida totalmente na janela de consulta, ou se intercepta mas não está contida, ou se a janela de consulta e a partição são disjuntas. Se a partição está totalmente contida na janela de consulta, todos os pontos nela contidos são reportados. Caso apenas intercepte, faz-se uma busca para verificar quais pontos pertencem à janela. E finalmente, se uma partição e a janela de consulta são disjuntas, a partição é descartada.

### 2.4.5 A Filter Tree

A Filter Tree [SK96] é um método de acesso idealizado para a execução eficiente de junções espaciais. Este método baseia-se nos seguintes princípios:

- representação hierárquica do espaço;
- separação dos objetos por tamanho;
- localidade de acessos.

[2] O complemento da partição $P$, determinado invertendo-se o bit menos significativo de $P$.




| | A | | B | | C | |
|---|---|---|---|---|---|---|
| $x_1$ | .55 | .10001100 | .25 | .01000000 | .45 | .01110011 |
| $x_2$ | .80 | .11001100 | .30 | .01001100 | .55 | .10001100 |
| $y_1$ | .15 | .00100110 | .65 | .10100110 | .90 | .11100110 |
| $y_2$ | .40 | .01100110 | .70 | .10110011 | .95 | .11110011 |
| Nível | 1 | | 3 | | 0 | |

Figura 2.17: Retângulos indexados em uma Filter Tree, com suas coordenadas e o cálculo dos níveis a que pertencem.

Os objetos são representados por meio de MBRs e o espaço onde se encontram os MBRs é mapeado para outro espaço, cujas dimensões variam de $[0, 1)^2$. Desta forma, as coordenadas dos MBRs e das janelas de consulta devem também ser transformadas na mesma proporção.

Este método divide o espaço através de uma hierarquia de grades regulares. A hieraquia possui $L + 1$ níveis, variando de 0 a $L$, e em dado nível $j$ a grade possui $4^j$ células de tamanho $\frac{1}{2^j} \times \frac{1}{2^j}$.

Cada objeto é associado ao nível mais baixo o qual ele pode ser completamente sobreposto por uma única célula. Dado um objeto, cujo MBR é representado pelos cantos inferior esquerdo $(x_1, y_1)$ e superior direito $(x_2, y_2)$, calcula-se o nível a que ele pertence a partir das representações binárias de suas coordenadas. Seja $b_x$ o número de bits mais significativos comuns a $x_1$ e $x_2$ e $b_y$ o número de bits mais significativos comuns a $y_1$ e $y_2$. Então $n = min(b_x, b_y)$ é igual ao nível ao qual o objeto deve ser associado.

Com este esquema, os retângulos maiores são associados aos níveis mais altos da hierarquia enquanto os menores tendem a ser associados aos níveis mais inferiores. No entanto, há a possibilidade de retângulos pequenos poderem ser associados aos níveis superiores, o que acontece quando esses retângulos interceptam as linhas de alguma grade logo nos primeiros níveis. Estas situações podem ser melhor visualizadas na Figura 2.17.

Para cada objeto, uma entrada denominada *descritor de entidade* é armazenada em um arquivo. Cada descritor de entidade possui as seguintes informações:

- as coordenadas do MBR do objeto;
- a coordenada linear na curva de Hilbert associada ao centro do MBR;
- um ponteiro para a página de disco onde estão os dados completos do objeto.

O arquivo que contém os descritores está organizado da seguinte forma:

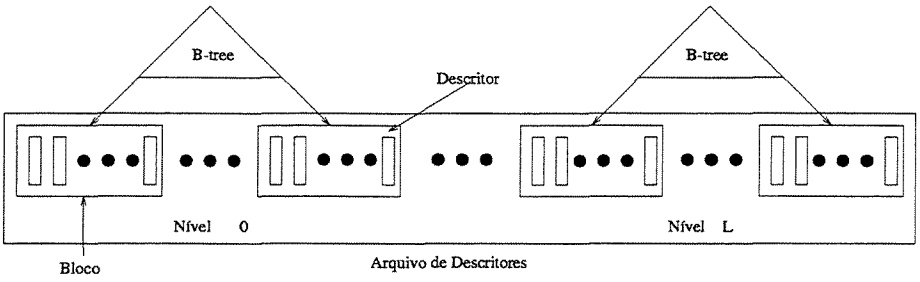



Figura 2.18: Representação da Estrutura de uma Filter Tree.

- os descritores de todos os objetos associados a um determinado nível são armazenados juntos;
- os descritores de cada nível são ordenados pelas coordenadas lineares dos pontos centrais dos MBRs correspondentes, segundo uma curva de Hilbert. Isto possibilita o armazenamento conjunto dos descritores de objetos pertencentes a uma mesma célula, visando a otimização dos acessos a disco;
- os descritores são armazenados em blocos;
- para cada nível da hierarquia há um índice de células (uma B-tree) que armazena a coordenada linear do último descritor em cada bloco.

Esta descrição pode ser compreendida através da Figura 2.18, uma representação da estrutura de uma Filter Tree.

A indexação de pontos pode ser problemática para a Filter Tree. Isto se deve ao fato de que, dependendo de sua localização, os pontos podem jamais interceptar as linhas de uma grade. Além disso, teoricamente não existe um tamanho de célula no qual um ponto não possa estar contido. Embora os pontos sejam representados por um número limitado de bits, de forma que sempre há um nível a ser associado a cada ponto, o tratamento de pontos pode aumentar muito o número de níveis. Para solucionar este problema, os autores sugerem a incorporação de mais um nível, denominado *nível infinito*. Sobre este nível recairiam todos os pontos que não estivessem sobre as linhas de alguma grade nos níveis superiores.

A busca por um objeto na Filter Tree é feita através da identificação do nível em que o objeto se encontra, e a subseqüente busca pelo bloco que contém o descritor do objeto na árvore B a partir da coordenada linear o ponto central do MBR do objeto. Obtendo o descritor do objeto, temos o ponteiro para a área de dados onde está o objeto.

A execução de *range queries* é feita verificando-se, em cada nível, todas as células que interceptam a janela de consulta. Dentro de cada nível, o conjunto de células a ser

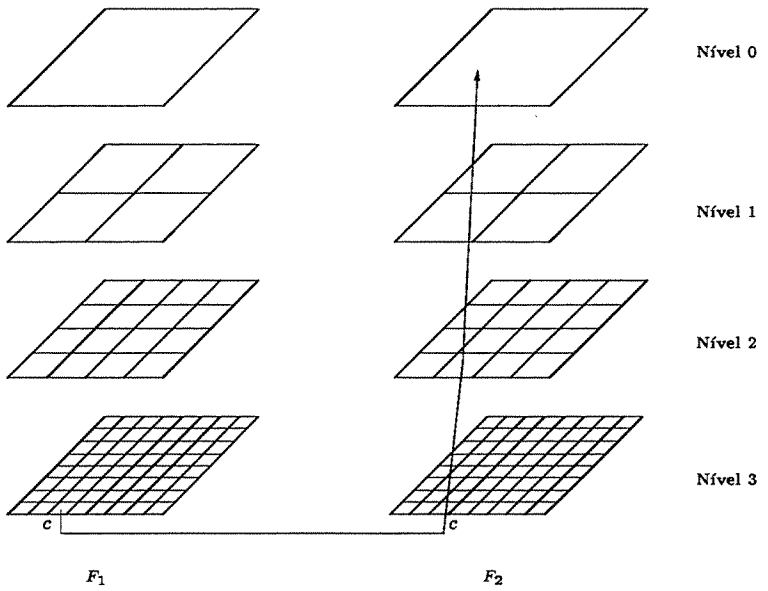



Figura 2.19: Processo de junção espacial de duas Filter Trees.

examinadas forma um conjunto de intervalos na curva de Hilbert. Este intervalo pode então ser utilizado para identificar o primeiro e o último bloco do descritor de identidade que contém os valores da curva de Hilbert do intervalo.

Os autores ressaltam que esta abordagem apresenta problemas em níveis muito baixos da hieraquia, uma vez que o número de células que interceptam a janela de consulta tende a ficar muito grande, gerando um grande número de intervalos. A saída é escolher um nível, denominado *containment level* ($c$), fazendo com que o conjunto de intervalos gerados nesse nível seja usado nos níveis subseqüentes. A escolha de valores grandes para $c$ faz com que menos áreas fora da janela de consulta sejam escolhidas ao custo de um maior número de células localizadas no limite da janela de consulta. Os autores fazem uma análise dos valores de $c$, ressaltando que a escolha apropriada de $c$ depende mais do número de entidades armazenadas na Filter Tree do que do tamanho da janela de consulta.

A junção espacial entre duas Filter Trees envolve um processo de varredura de índices. Entretanto, a Filter Tree foi projetada de forma que, para quaisquer pares de conjuntos de dados, a junção espacial demande uma quantia mínima de operações de entrada/saída (I/O). Como exemplo, considere a Figura 2.19, com duas hierarquias $F_1$ e $F_2$, com quatro níveis cada. Se quisermos encontrar os elementos de $F_2$ que interceptam os elementos da célula $c$ de $F_1$, temos que percorrer apenas a célula $c$ do mesmo nível em $F_2$ e as células dos níveis superiores que a contém.

Os *intervalos de processamento* de junção são identificados através dos marcadores finais tomados de cada bloco do arquivo de descritores de identidade. Seja $e_{l_j}^F$ o maior valor de Hilbert de qualquer descritor de entidade no $j$-ésimo bloco do nível $l$ da Filter Tree $F$. Então há tantos marcadores finais quanto blocos no arquivo de descritores de ambas as árvores. O conjunto de $e_{l_j}^F$ valores é classificado e valores duplicados são removidos. Os valores de Hilbert, em pares sucessivos de valores de marcadores na lista classificada, têm a propriedade de que estão totalmente contidos dentro de um bloco em cada nível de cada árvore participante. Desta forma é possível processar cada intervalo de uma vez enquanto se mantém na memória apenas um bloco de cada nível da árvore. Quando o processamento de todos os pares chegar ao valor de Hilbert igual a $e_{l_j}^{F_i}$ significa que terminamos com o $j$-ésimo bloco do nível $l$ da árvore $F_i$, e o substituímos pelo bloco $j+1$ do nível $l$, e processamos o intervalo seguinte. Os $e_{l_j}^{F_i}$ valores não são únicos, o que significa que intervalos terminados por valores não únicos causarão a leitura de um mesmo bloco mais de uma vez.

Dentro de um intervalo de processamento, as ações realizadas são as seguintes: os níveis de 0 a $L$ de cada árvore são endereçados, e o passo da junção ilustrado na Figura 2.19 é realizado em todo o conjunto de entidades no bloco daquele nível. Seja $S_l^{F_i}(e_n, e_{n+1})$ o conjunto de entidades no nível $l$ da árvore $F_i$, cujos valores de Hilbert estão no intervalo $(e_n, e_{n+1})$. Os intervalos de processamento estão definidos de forma que todos os descritores de identidade contidos neste conjunto estarão na memória quando o intervalo for processado. Assim, para os níveis de $l = 0, \ldots, L$, são combinadas:

- os objetos em $S_l^{F_1}(e_n, e_{n+1})$ em relação àqueles de $S_{l-i}^{F_2}(e_n, e_{n+1})$, para $i = 0, \ldots, l$;
- os objetos $S_l^{F_2}(e_n, e_{n+1})$ em relação àqueles em $S_{l-i}^{F_1}(e_n, e_{n+1})$, para $i = 1, \ldots, l$.

Os intervalos de $i$ diferem para evitar que $S_{l-i}^{F_2}(e_n, e_{n+1})$ e $S_{l-i}^{F_1}(e_n, e_{n+1})$ sejam combinados duas vezes.

## 2.5 Métodos baseados em árvores binárias

Os métodos derivados de árvores binárias, que estendem as características básicas dessas estruturas para o espaço $k$-dimensional, dividem o espaço de forma recursiva e hierárquica em sub-regiões disjuntas. Entretanto, este tipo de método herda também características indesejáveis das árvores binárias, o que as tornam inapropriadas à utilização em sistemas gerenciadores de bancos de dados. Estas características são as seguintes:

- árvores binárias não são balanceadas, pois a altura da árvore é determinada pela distribuição dos dados e pela ordem de inserção dos dados na árvore. Isto faz com

que consultas neste tipo de método tenham desempenho inferior ao das consultas em estruturas balanceadas. Embora existam algoritmos para o balanceamento de árvores binárias e suas extensões para espaços $k$-dimensionais, sua execução depende do conhecimento prévio dos dados;

- árvores binárias são estáticas, o que faz com que seu desempenho se degrade com a inclusão e exclusão de dados da estrutura. Mesmo que inicialmente a árvore esteja balanceada, com sucessivas remoções, a árvore pode degenerar-se a uma lista ligada, prejudicando o desempenho das consultas. Uma solução seria a reorganização periódica do índice, o que é inaceitável sob determinadas situações pois indisponibiliza o índice;

- árvores binárias possuem baixo *fan-out*, tornado-as inapropriadas para a indexação de dados em disco. Em geral, as entradas que são armazenadas em estruturas espaciais têm poucos *bytes* se comparadas ao tamanho de uma página de disco, fazendo com que, se armazenarmos apenas um nó em cada página, teremos grande desperdício de espaço. Uma solução é armazenar vários nós em uma mesma página.

A seguir apresentaremos dois métodos baseados em árvores binárias desenvolvidos para a indexação de pontos, a Point Quadtree e a $k$-$d$-tree.

### 2.5.1 Point Quadtree

A Point Quadtree [FB74] é um método criado para indexação de pontos em um espaço $k$-dimensional. Ela baseia-se na decomposição recursiva do espaço em quatro sub-regiões denominadas "quadrantes". Por simplicidade, consideraremos como exemplos apenas espaços bidimensionais.

Neste método, cada ponto inserido determina a posição das linhas que dividem o espaço. Estas linhas particionam o espaço em todas as dimensões, decompondo o espaço hierarquicamente, ou seja, a divisão feita pela inserção de um ponto é restrita ao subespaço que o contém.

Cada nó da árvore armazena as seguintes informações:

- as coordenadas de um único ponto, que serve para a subdivisão do espaço;

- um apontador para o registro de dados associado ao ponto;

- apontadores para quatro sub-árvores que representam os quatro quadrantes em que o espaço é divido. Estes quadrantes são denominados nordeste, noroeste, sudoeste e sudeste, que correspondem às sub-árvores 1, 2, 3 e 4, respectivamente.

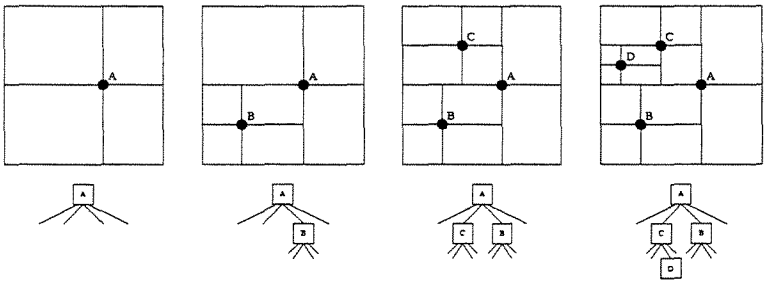



Figura 2.20: Processo de inserção de pontos em uma Point Quadtree.

Em um espaço $k$-dimensional, cada nó da estrutura tem $2^k$ filhos, que correspondem às $2^k$ regiões em que o espaço é particionado. Os autores consideram que não há dois pontos diferentes com as mesmas coordenadas. Entretanto, em casos nos quais colisões são permitidas, os autores sugerem que o apontador para o registro de dados referencie alguma estrutura que armazene os pontos cujas coordenadas coincidam.

A Figura 2.20 mostra o processo de crescimento e particionamento da árvore. A inserção de pontos da Quadtree inicia-se com a inserção do primeiro ponto na raiz. As coordenadas deste primeiro ponto servirão como origem para a primeira divisão do espaço. Novos pontos são inseridos percorrendo-se a árvore de acordo com o posicionamento destes em relação aos quadrantes de cada nó da árvore. A cada ponto inserido, uma nova subdivisão do espaço é feita na região onde foi inserido.

Range queries são executadas em dois passos: verificar se o ponto relacionado ao nó corrente pertence à janela de consulta e reportá-lo na consulta em caso afirmativo; recursivamente chamar o procedimento em cada sub-árvore não vazia cujo quadrante intercepte a janela de consulta.

A consulta para verificação de pontos na estrutura é similar ao processo de inserção.

A remoção de elementos proposta em [FB74] faz a reinserção de todos os nós pertencentes às sub-árvores do nó excluído. Isto pode ser muito ineficiente, dependendo da posição em que se encontra o ponto. Um algoritmo alternativo procura minimizar a quantidade de nós a serem reinseridos através da substituição do nó escolhido por seu descendente mais próximo.

### 2.5.2 *k-d* Tree

A *k-d* Tree [Ben75] é um método de acesso a pontos de um espaço $k$-dimensional que, da mesma forma que a Point Quadtree, faz decomposição recursiva do espaço orientada pela posição de inserção dos pontos, restrita ao sub-espaço que o contém. Entretanto,

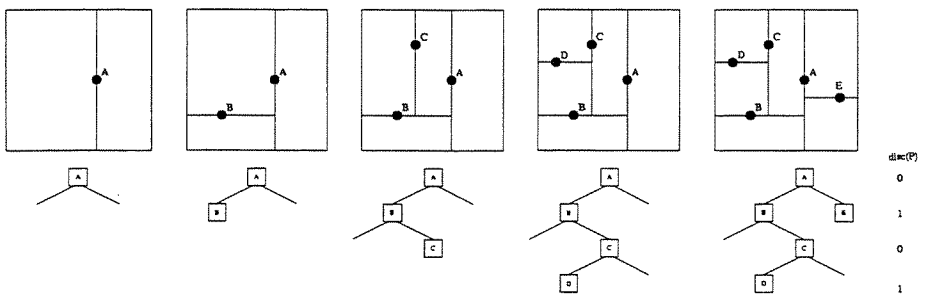


Figura 2.21: Processo de inserção de pontos em uma *k-d* Tree. A sub-árvore esquerda representa $LOSON(P)$ e a sub-árvore direita representa $HISON(P)$.

enquanto que na Quad Tree a divisão é feita em todas as dimensões, na *k-d* Tree a divisão é feita em apenas uma das dimensões do espaço. Ou seja, cada ponto inserido divide sua sub-região em duas novas regiões.

Cada nó da estrutura é composto dos seguintes elementos:

- as $k$ coordenadas de um ponto $P$ (como na Quad Tree, cada nó armazena apenas um ponto);
- um apontador para o registro de dados referente a $P$ (como na Point Quadtree, não se admite que dois pontos tenham as mesmas coordenadas, mas admiti-se uma solução parecida com a adotada na Point Quadtree);
- um discriminador $disc(P)$, que indica em qual dimensão o espaço está sendo particionado naquele nível da árvore;
- dois apontadores para suas sub-árvores, a saber
  - $LOSON(P)$, que referencia a sub-árvore cujos elementos possuem coordenada na dimensão determinada por $disc(P)$ menores do $P$;
  - $HISON(P)$, que referencia a sub-árvore cujos elementos possuem coordenada na dimensão determinada por $disc(P)$ maiores do $P$;

Para se inserir um novo dado na estrutura, percorre-se a árvore comparando a coordenada do ponto a ser inserido com a do ponto armazenado em cada nó, na dimensão dada pelo seu discriminador $disc(P)$. A Figura 2.21 apresenta o processo de crescimento de uma *k-d* Tree.

Consultas para verficação da existência de um ponto na estrutura são feitas de forma parecida com o processo de inserção: percorre-se a árvore a partir da raiz até encontrar

o dado desejado ou encontrar um ponteiro nulo. Consultas de *range queries* são feitas de forma que, a cada nível, apenas as sub-árvores que representam cada sub-espaço que interceptam a janela de consulta sejam percorridas. A cada nó visitado, o algoritmo verifica se o ponto que ele armazena intercepta a janela de consulta.

A exclusão de um nó cujas sub-árvores são vazias pode ser feita diretamente. Entretanto, ao se remover um nó $P$ que possui descendentes, deve-se substituí-lo por um de seus descendentes $Q$. O nó $Q$ é escolhido de forma que todos os filhos de $P$ presentes na sub-árvore apontada por $LOSON(P)$ também estejam na sub-árvore apontada por $LOSON(Q)$ e todos os que estavam na árvore apontada por $HISON(P)$ também estejam na sub-árvore apontada por $HISON(Q)$. Para que estas condições sejam satisfeitas, $Q$ deve ser o elemento mais próximo possível de $P$ na dimensão determinada por $disc(P)$. A substituição de $P$ por $Q$ implica na exclusão de $Q$ da posição onde se encontrava. Assim, se $Q$ tiver descendentes, esta operação deverá ser recursivamente executada até encontrar um elemento que não possua descendentes.

## 2.6 Métodos derivados de estruturas *hash*

As organizações de arquivos que utilizam *hashing* aplicam uma função sobre os valores de um ou mais atributos de cada registro (chamados *chaves de randomização*), que determina o endereço do bloco (*bucket*) onde deverá ser armazenado o registro.

Porém, neste tipo de endereçamento, o número de blocos deve ser mantido fixo, pois é um dos parâmetros da função de randomização. Algumas técnicas de *hashing* permitem a expansão e redução dinâmica do arquivo através da associação entre o valor da chave de randomização e o endereço do bloco feita por intermédio de um *diretório*, estrutura implementada através de um vetor ou uma árvore binária. Nestes casos, a função de *hashing* é utilizada para o cálculo da entrada correspondente no diretório, que por sua vez contém o endereço apropriado. Ao se ultrapassar a capacidade de armazenamento de um bloco, este é divido em dois e o diretório é reorganizado para refletir a mudança. Analogamente, caso dois blocos tenham ocupação menor que o mínimo exigido, estes podem ser substituídos por um único bloco.

Os métodos de acesso espaciais derivados de estruturas *hash* utilizam as coordenadas espacias dos objetos como chave de randomização, preocupando-se em mapear objetos espacialmente próximos para um mesmo bloco ou blocos adjacentes, otimizando a execução de *range queries*. Nesta seção apresentaremos a proposta do Grid File [NHS84].

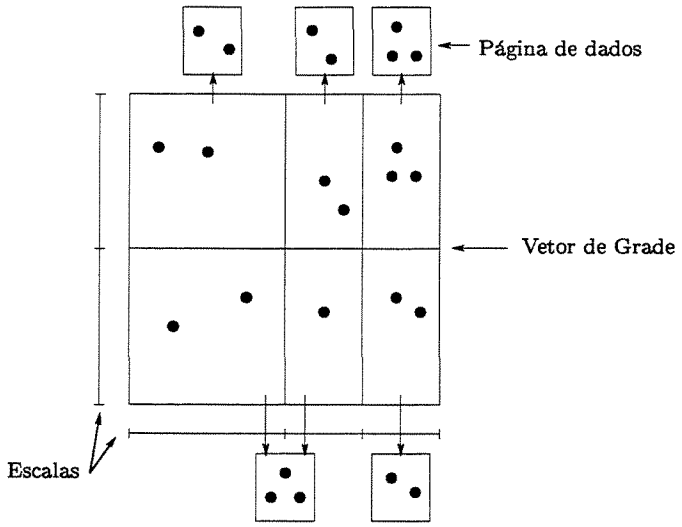



Figura 2.22: Estrutura de um Grid File em um espaço bidimensional, com número máximo de elementos por blocos igual a três.

### 2.6.1 Grid File

O Grid File, de Nievergelt, Hinterberger e Sevcik [NHS84], é um método de acesso a pontos que busca a execução de *point queries* em um espaço $K$-dimensional com apenas dois acessos a disco e a execução eficiente de *ranges queries*. Este método divide o espaço em células hiper-retangulares, denominadas *grid blocks*, através da inserção de hiperplanos $(k-1)$-dimensionais paralelos aos eixos das dimensões e associa cada célula a um bloco (página de dados) ou *bucket* de disco através de um diretório.

Um *diretório* é composto por dois tipos de estruturas:

- um vetor $k$-dimensional denominado *vetor de grade*, cujos elementos correspondem às células nas quais o espaço foi particionado. Cada entrada do diretório mantém o endereço de um bloco ou *bucket* de disco onde estão armazenados os pontos da célula correspondente. Para evitar sub-utilização de blocos de disco, duas ou mais células podem referenciar um mesmo bloco ou *bucket*, desde que a soma de seus elementos não ultrapasse o número máximo de objetos permitido;

- $k$ vetores unidimensionais denominadas *escalas*, sendo que cada vetor armazena a posição dos hiperplanos que particionam o espaço $S$ na dimensão $k$.

A Figura 2.22 mostra a organização de um Grid File com capacidade máxima de armazenamento de três pontos por bloco em um espaço bidimensional.

### Inserção

A identificação do bloco onde um ponto $P$ deve ser inserido é feita através da comparação das coordenadas de inserção dos hiperplanos armazenadas nas escalas para determinar que célula contém $P$. A entrada correspondente do diretório é lida do disco, obtendo assim o endereço do bloco de disco $B$, que é então recuperado.

Se o bloco ainda não atingiu sua capacidade máxima de armazenamento, entramos no caso trivial. Caso contrário, a região correspondente a $B$ deve ser dividida em duas sub-regiões e cada região deve ser associada a um bloco diferente. Ao dividirmos uma região em duas, dois casos podem ocorrer:

- se a região correspondente a $B$ contém apenas uma célula, então é necessário inserir um novo hiperplano, particionando a região. A escolha da dimensão onde será inserido o novo hiperplano é feita ciclicamente, ou seja, se no particionamento anterior o espaço foi dividido na dimensão $d$, então o espaço será particionado na dimensão $d+1$. A inserção de um novo hiperplano na dimensão $d$ implica na atualização do vetor de escala daquela dimensão;

- caso a região correspondente a $B$ tenha mais de uma célula, então pode-se deduzir que há pelo menos um hiperplano que a corta, fazendo com que a inserção de um novo hiperplano seja desnecessária. Deve-se apenas alocar um novo bloco de disco e redistribuir os pontos de acordo com o hiperplano pré-existente.

A Figura 2.23 ilustra esta situação. A inserção do ponto $p_1$ faz com que o bloco $B$ ultrapasse sua capacidade máxima, que armazena os pontos de apenas uma célula. Faz-se necessária o particionamento da região, o que é feito paralelamente em relação ao eixo vertical. Note que o particionamento da região armazenada em $B$ faz com que a região armazenada em $A$ também seja particionada, fazendo com que $A$ armazene os pontos de duas células. A inserção do ponto $p_2$ não exige que sua região seja particionada porque esta ainda não havia ultrapassado seu limite máximo de entrada, mas é necessária a alocação de um novo bloco, pois o bloco $C$, que armazenava pontos de duas células, atingiu sua capacidade máxima.

Nievergelt, Hinterberger e Sevcik sugerem que todas as células tenham um só tamanho para permitir o cálculo do endereço no disco. Esta proposta, entretanto, faz com que, cada vez que uma célula seja dividida por um hiperplano em uma dimensão, todas as outras também seja divididas na mesma dimensão. Desta forma, todas as vezes que se fizer um *split* o diretório dobra de tamanho. Isto é exemplificado na Figura 2.24: quando o bloco $B$ sofre um *split*, sua região é dividida em duas através da inserção de um hiperplano paralelo ao eixo vertical, que acarreta uma divisão das células da região do bloco $C$ para manter a uniformidade no tamanho das células.

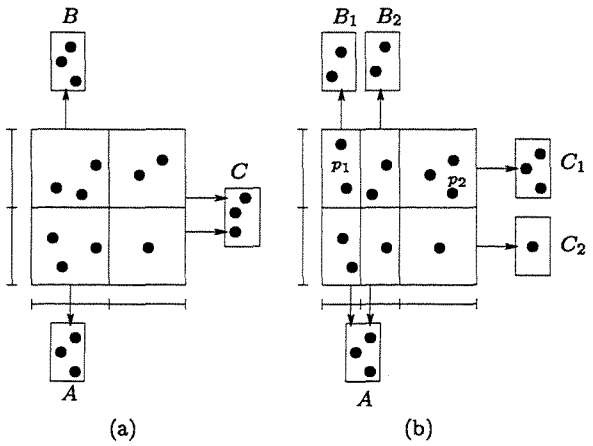



Figura 2.23: Tratamento de *overflow* em um Grid File. O espaço inicial é representado na Figura (a) enquanto que a Figura (b) representa o espaço após a inserção dos pontos $p_1$ e $p_2$.

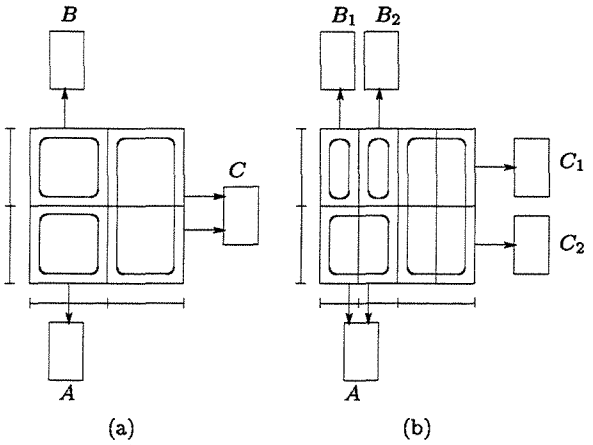



Figura 2.24: Um Grid File com tamanho de células fixo.

### Consulta

A verificação da existência de um ponto na estrutura é feita da mesma forma que na determinação do bloco em que um ponto deve ser incluído e, como na inserção, apenas dois acessos a disco são necessários.

A execução de *Range queries* é feita através da identificação, por meio das escalas, das células que interceptam a janela de consulta, seguida da recuperação das entradas do diretório e das páginas de dados correspondentes. Células que estão totalmente contidas na janela de consulta têm todos os seus pontos reportados, enquanto que, em células que interceptam parcialmente a janela de consulta, todos os pontos devem ser verificados.

### Remoção

Ao remover um ponto, o bloco onde ele se encontrava pode sofrer uma junção com outro bloco, caso tenha menos elementos que o mínimo permitido. Para que esta junção aconteça, dois requisitos devem ser satisfeitos:

- a nova região a ser formada pela junção dos blocos deve ser um hiper-retângulo;
- a soma do número de pontos armazenados nos dois blocos não deve ultrapassar um limite pré-determinado. A intenção é não criar uma região que logo teria que ser particionada.

Inicialmente, um bloco pode sofrer uma junção com qualquer bloco associado a uma região vizinha a sua desde que os requisitos apresentados seja satisfeitos. Os autores apresentam duas políticas para a escolha dos blocos candidatos à junção:

- *neighbor system*, na qual o bloco pode sofrer uma junção com qualquer dos seus vizinhos nas $k$ dimensões;
- *buddy system*, na qual um bloco só pode ser unido a um bloco específico em cada dimensão – seu *buddy*. Dois blocos são considerados *buddies* um do outro se suas regiões podem ser obtidas pela divisão de uma região superior ao meio. Pode-se notar que os elementos de um *buddy system* são um subconjunto de um *neighbor system*.

A Figura 2.25 mostra algumas junções possíveis em cada sistema, apresentando o *buddy system* como subconjunto do *neighbor system*.

A junção de dois blocos torna desnecessária a exitência do hiperplano que separa suas regiões quando ele não separa nenhum outro par de regiões. A Figura 2.26 exemplifica esta situação. Quando o ponto $p_1$ é removido, região à qual $p_1$ pertencia atinge valores

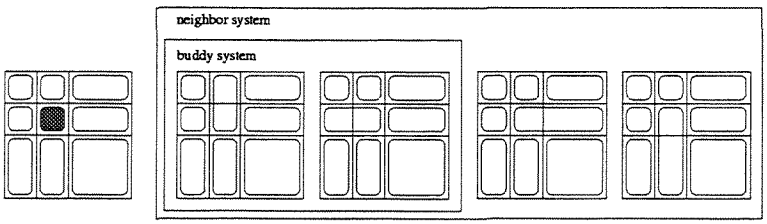



Figura 2.25: A região em destaque na figura mais à esquerda representa a célula que deverá sofrer a junção. As outras figuras mostram as possíveis células que podem ser unidas segundo as duas políticas propostas.

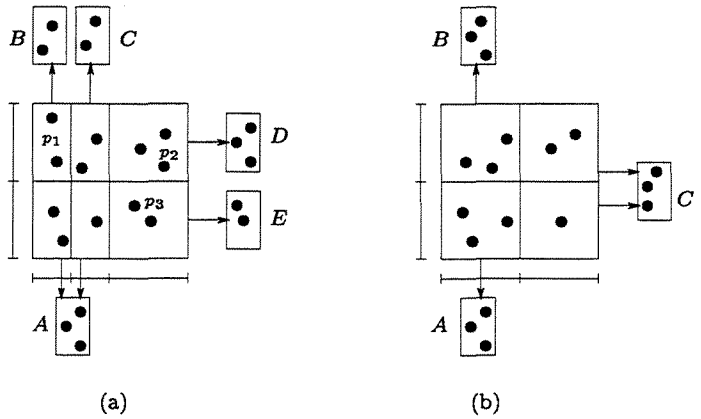



Figura 2.26: Exemplos de junção de duas regiões quando a ocupação dos blocos atinge valores menores do que o permitido.

menores do que o permitido. O que se faz então é remover o hiperplano que separa as regiões representadas por $B$ e $C$. Esta remoção é possível por que as duas células adjacentes às regiões recém unidas pertencem a uma mesma região. Deve-se também ajustar o vetor de escala e o vetor de grade. Por outro lado, a remoção dos pontos $p_2$ e $p_3$, apesar de causar a junção de $D$ e $E$, não causa a remoção do hiperplano que as separava, pois este ainda separa as regiões dos blocos $A$ e $B$.

Se o diretório é implementado de forma que todas as células tenham o mesmo tamanho, a retirada do hiperplano que corta a região que sofre a junção só é possível se toda a grade puder ser reorganizada de forma que o novo comprimento de intervalo seja usado na escala correspondente.

#### Indexação de objetos de dimensão maior que zero

O Grid File pode ser utilizado na indexação de objetos de dimensão maior do que zero. Para isto, deve-se transformar seus MBRs em pontos em um espaço $2k$-dimensional, ou ainda, utilizar a técnica de *abstração*, apresentadas na seção 2.3.1.

## 2.7 Métodos baseados em árvores multiárias

Árvores multiárias caracterizam-se pela capacidade que seus nós possuem de armazenar vários filhos, em contraposição às árvores binárias. Esta característica torna-as apropriadas ao gerenciamento de dados em memória secundária, uma vez que permite que um nó da árvore seja associado a uma página do disco, bastando que o número máximo de entradas de cada nó esteja relacionado ao tamanho da página e ao tamanho de cada entrada do nó. Associadas a esta característica, mais duas contribuíram para que árvores multiárias fossem amplamente utilizadas na indexação de dados:

- são balanceadas, isto é, todas as folhas aparecem no mesmo nível. Esta característica permite que, mesmo em distribuições irregulares de dados, consultas sejam realizadas em tempo logarítmico;
- são dinâmicas, ou seja, seu desempenho não se degrada devido às constantes atualizações nos dados, tornando desnecessárias reorganizações periódicas.

Os principais métodos baseados em árvores multiárias pertencem à família das R-trees, uma generalização das $B^+$-trees para espaços $k$-dimensionais. Discutiremos alguns métodos desta família nas seções seguintes.

### 2.7.1 R-trees

A R-tree, proposta por Guttman [Gut84], é um método que agrupa os MBRs dos objetos de um espaço $k$-dimensional em uma hierarquia de MBRs. A árvore possui dois tipos de nós:

- nós folhas, que armazenam informações sobre os objetos indexados na forma (*identificador, MBR*), onde o primeiro campo é o identificador do objeto e o segundo campo é o MBR do objeto;
- nós não-folha, que contém informações da forma (*ponteiro, MBR*), onde o primeiro campo é um ponteiro para um nó de um nível inferior na hierarquia e MBR é o retângulo envolvente mínimo que envolve todos os MBRs dos nós subordinados.

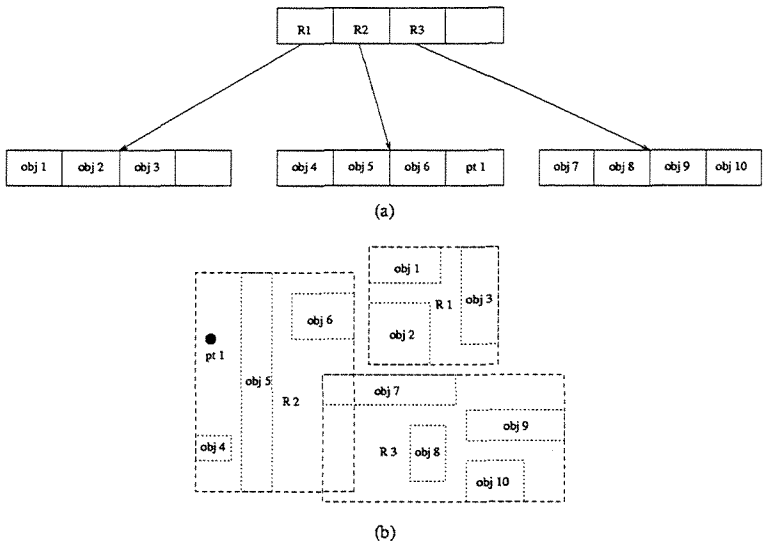



Figura 2.27: Uma R-tree (a) e os retângulos e pontos indexados (b).

Cada nó é armazenado em uma página de disco, cuja capacidade máxima é $M$ entradas do tipo (*ponteiro, MBR*) ou (*identificador, MBR*). O número mínimo de entradas em um nó é dado por $m \leq \frac{M}{2}$. Definidos estes dois valores, um nó deve respeitar as seguintes características:

- um nó não deve possuir mais do que $M$ e menos do que $m$ entradas, a menos que seja raiz;
- para cada registro *(identificador, MBR)* em um nó folha, MBR é o menor retângulo que espacialmente contém o objeto de dados representado pelo registro;
- todo nó não-folha possui entre $m$ e $M$ filhos, a menos que seja raiz;
- para cada entrada *(ponteiro, MBR)* em um nó não-folha, MBR é o menor retângulo que espacialmente contém os retângulos do nó filho apontado por *ponteiro*;
- o nó raiz possui pelo menos dois filhos, a menos que seja folha;
- todas as folhas aparecem no mesmo nível.

A Figura 2.27 mostra os pontos e retângulos indexados por uma R-tree. Note que os MBRs de um mesmo nível podem se interceptar.

### Inserção

Como em uma $B^+$-tree, um novo objeto é inserido nas folhas da árvore. Para inserir um novo objeto, percorre-se a estrutura da árvore a partir da raiz e segue-se o ponteiro cujo MBR associado necessite do menor aumento de área para conter o MBR do objeto. Casos de empate são resolvidos escolhendo-se a entrada cujo MBR possuir a menor área.

Caso o número de elementos contidos no nó seja menor que $M$, o objeto é inserido e, se necessário, os MBRs dos nós superiores na hierarquia são ajustados. Entretanto, caso a folha $i$ já tenha atingido o número máximo de entradas permitido (fato conhecido como *overflow*), deve sofrer uma operação de *split*, ou seja, uma nova folha $j$ é criada e as entradas da folha que sofre o *split* mais a nova entrada são divididas entre $i$ e $j$. A operação de *split* pode se propagar até os níveis superiores da estrutura, atingindo até mesmo a raiz, uma vez que a criação da folha $j$ implica na criação de uma nova entrada no nó imediatamente superior a $i$, que pode sofrer *overflow* e assim sucessivamente.

A operação de *split* busca dividir as entradas de tal forma que os MBRs dos nós que sofreram o *split* possuam a menor área possível. Da mesma forma como na rotina que escolhe a folha onde será feita uma inclusão, o objetivo é tentar minimizar as áreas dos MBRs que envolvem as entradas de cada nó, minimizando assim o número de nós visitados em uma consulta. Isto se deve ao fato de que, quanto menor um MBR, menor é sua chance de interceptar uma janela de consulta. Com a diminuição da área dos MBRs, diminui-se também o *dead space*, um dos responsáveis pela detecção de *false hits*, aumentando a eficiência de consultas.

Guttman propõe três algoritmos diferentes para a realização do *split*:

- um exaustivo, que encontra a melhor divisão possível, mas possui custo exponencial quanto ao número de entradas, sendo proibitivo para valores realísticos de $M$;
- um quadrático, que escolhe entre as $M + 1$ entradas as duas que resultariam na maior área se fossem inseridas no mesmo grupo, e as separa como sementes de dois novos grupos. As entradas restantes são tomadas uma a uma e atribuídas ao grupo em que causaria menor aumento de área, e no caso de empate, no grupo com menor área;
- um linear, que escolhe como sementes os dois MBRs que apresentem a maior distância normalizada entre si. Em cada dimensão, encontra-se a entrada cujo MBR possua a maior coordenada mínima e a entrada cujo MBR possua a menor coordenada máxima e normaliza-se essa separação dividindo-a pela extensão do conjunto inteiro naquela dimensão. As sementes candidatas escolhidas são aquelas que apresentam a maior separação normalizada em qualquer dimensão. Após escolhidas as sementes, aleatórias para cada grupo, procede-se como no algoritmo quadrático.

Greene [Gre89] propõe uma implementação que utiliza outro método de *split* quadrático. Após a escolha das sementes do dois grupos, que é feito da mesma forma como no método quadrático de Guttman, determina-se em que dimensão se dará a partição. A dimensão a ser particionada é determinada computando-se a maior distância normalizada entre os MBRs das sementes. A entradas são então classificadas pela menor coordenada de seus MBRs nessa dimensão e distribuem-se as $(M + 1)/2$ primeiras entradas a um grupo e as restantes ao outro grupo.

Ang e Tan [AT97] sugerem um novo algoritmo linear para realização de *split* que busca, através da minimização da área de sobreposição, otimizar o desempenho nas buscas em R-trees. Em um espaço bidimensional, um retângulo $r$ pode ser representado por uma 4-tupla ($x_1$, $y_1$, $x_2$, $y_2$) formada por dois pares de coordenadas, a saber: coordenadas do vértice inferior esquerdo ($x_1$, $y_1$) e coordenadas do vértice superior direito ($x_2$, $y_2$). Seja $R_0 = (L, B, R, T)$ o MBR do nó que sofreu o *overflow*. O método utiliza quatro listas $LIST_L$, $LIST_B$, $LIST_R$ e $LIST_T$ para armazenar aqueles retângulos que estão mais próximos de uma borda do que da oposta.

O algoritmo funciona da seguinte forma: para todo retângulo $r = (x_l, y_l, x_h, y_h)$ do nó que sofreu *overflow*, verifica-se se $x_h - L < R - x_l$, inserindo $r$ em $LIST_L$ ou, caso contrário, inserindo $r$ em $LIST_R$. Verifica-se também se $y_h - B < T - y_l$, inserindo $r$ em $LIST_B$ ou, caso contrário inserindo $r$ em $LIST_T$. Ao término dos testes, se $max(|LIST_L|, |LIST_R|) < max(|LIST_B|, |LIST_T|)$, então divide-se o nó ao longo do eixo $x$, caso contrário, divide-se ao longo do eixo $y$.

Segundo experimentos feitos pelos autores, este algoritmo teve desempenho muito superior nas realizações de *split* além de melhorar o desempenho de consultas executadas após a realização de *split*.

Em [GLL98], García, López e Leutenegger apresentam um algoritmo de complexidade $O(n^k)$ para se encontrar a partição ótima de um nó em espaços $k$-dimesionais. Os autores mostram ainda que, se não houver restrição quanto ao número de elementos de cada novo conjunto, apenas $O(n^2)$ pares de retângulos precisam ser verificados. Entretanto, em [Car98] mostrou-se que há um caso onde este algoritmo falha, e que não é tratada no trabalho original: se, no conjunto de retângulos a serem repartidos, um dos retângulos contém todos os outros, e nenhum destes toca a borda do primeiro, o algoritmo devolve uma partição vazia e a outra com todos os retângulos. Como este resultado não pode ser aceito quando há uma restrição de cardinalidade, a operação de *split* falha.

### Consulta

Consultas em uma R-tree são realizadas avaliando-se predicados sobre uma janela de consulta e os MBRs armazenados nos nós da árvore. Para se determinar quais objetos interceptam a janela de consulta, percorre-se a árvore a partir da raiz, descendo até

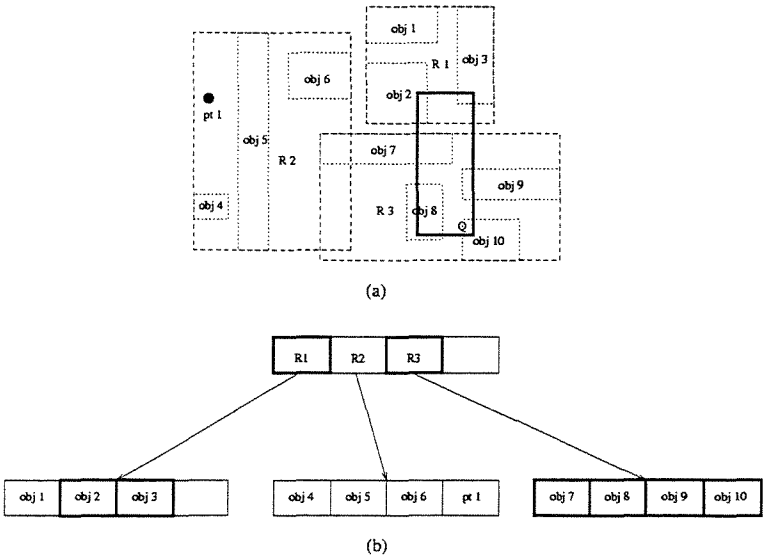



Figura 2.28: Consulta $Q$ sobre um conjunto de objetos (a) e os caminhos percorridos na árvore para a determinação da resposta (b).

as folhas, de modo similar a uma B-tree. Se o nó não é uma folha, verificam-se todas as entradas em busca daquelas cujo MBR intercepta a janela de consulta e, para toda entrada nessa condição, verifica-se o nó associado. Caso o nó seja folha, as entradas que interceptam a janela de consulta são computadas no conjunto resposta. Veja na Figura 2.28 um exemplo de consultas sobre um conjunto de retângulos e pontos juntamente com os caminhos percorridos na árvore para encontrar as respostas.

### Remoção

A remoção de uma entrada da árvore inicia-se com a determinação da folha onde se encontra a mesma, o que é feito chamando-se o procedimento de consulta. Então elimina-se a entrada.

Após eliminar a entrada, se o nó onde ela se encontrava contiver $m$ ou mais entradas, os MBRs dos nós pertencentes ao caminho percorrido até a folha devem ser verificados e se necessário, ajustados. Caso o número de entradas seja menor que o exigido (evento conhecido por *underflow*), as entradas restantes são atribuídas a um conjunto $Q$ para serem re-inseridas mais tarde. A remoção de um nó implica na remoção na entrada correspondente a esse nó em seu nó pai, o que faz com que o *underflow* possa se propagar até os níveis superiores da árvore, incluindo a raiz (o que acontece quando lhe resta apenas um filho). Assim, caso um nó não-folha sofra *underflow* suas entradas também são atribuídas ao conjunto $Q$. Ao se atingir um nível ao qual o *underflow* não se propagou, ajusta-se os MBRs até a raiz, se necessário.

Inicia-se então a reinserção das entradas contidas no conjunto $Q$. A reinserção é iniciada pelas entradas inseridas em $Q$ mais recentemente, que poderão guiar a reinserção das entradas remanescentes. Toda entrada é inserida no mesmo nível ao qual pertencia anteriormente, o que garante que todas as folhas estarão no mesmo nível.

O procedimento descrito acima difere da operação realizada em uma B-tree, embora uma abordagem parecida possa ser aplicada: as entradas restantes podem ser atribuídas ao nó "irmão" que tenha o menor aumento de área, ou podem ser distribuídas entre vários nós. Note que as duas alternativas podem causar *overflow* dos nós. Guttman justifica a utilização da reinserção por duas razões: primeiramente porque realiza a mesma função e é mais simples de se implementar, uma vez que a rotina de inserção pode ser reutilizada para este fim. O desempenho nesta operação não é afetado uma vez que as páginas visitadas durante a reinserção já estarão na memória, pois foram utilizadas durante a busca pela entrada. A outra razão é que a reinserção vai aos poucos refinando a estrutura espacial da árvore, evitando a deterioração gradual que poderia ocorrer se cada entrada estivesse sempre sob o mesmo nó pai.

### 2.7.2 R$^+$-tree

A R$^+$-tree, desenvolvida por [SRF87], é um método que procura dividir o espaço sem sobreposição dos MBRs de um mesmo nível. Com isso, busca eliminar a interseção de retângulos existente na R-tree e, em conseqüência, melhorar o desempenho de consultas. Para isso, o MBR de um objeto é decomposto em uma uma coleção de sub-retângulos disjuntos, cuja união é o próprio MBR do objeto, sempre que isso seja necessário para que os MBRs dos nós superiores não se interceptem.

Sua estrutura é a mesma das R-trees, mas as regras de construção são diferentes:

- para cada entrada (*ponteiro, MBR*) em um nó não-folha, a sub-árvore apontada por *ponteiro* contém um retângulo $R$ se, e somente se, $R$ for totalmente sobreposto por MBR. Se $R$ está em um nó folha, $R$ pode apenas interceptar MBR.
- para quaisquer duas entradas ($ponteiro_1$, $MBR_1$) e ($ponteiro_2$, $MBR_2$) de um nó não-folha, a interseção entre $MBR_1$ e $MBR_2$ é nula;
- a raiz possui pelo menos dois filhos, a menos que seja folha;
- todas as folhas se encontram no mesmo nível.

A Figura 2.29 mostra os pontos e retângulos indexados por uma R$^+$-tree. Note que o retângulo $obj_7$ foi representado nas folhas $R_2$ e $R_3$ para permitir que os retângulos envolventes fossem disjuntos.

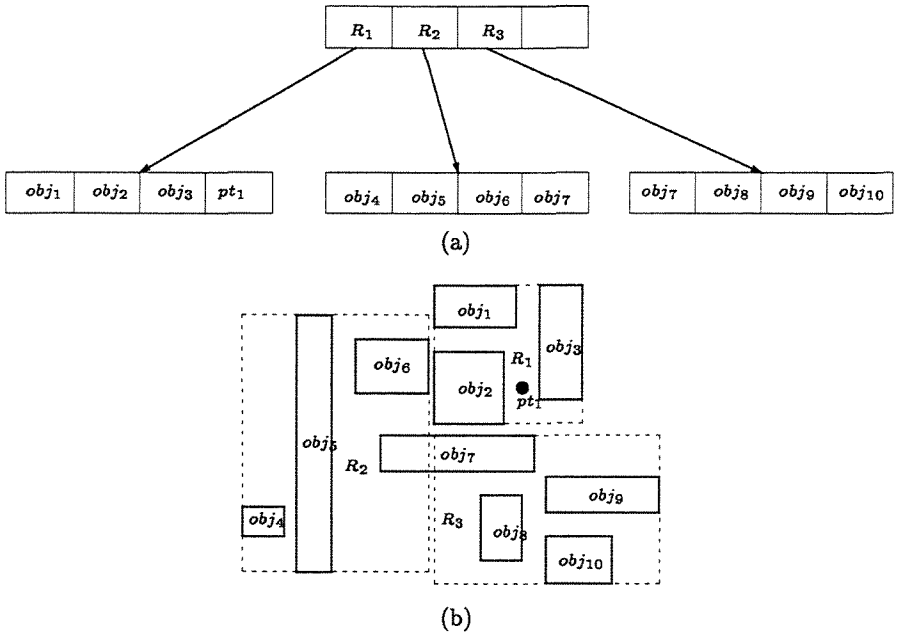



Figura 2.29: Uma $R^+$-tree (a) e os retângulos e pontos indexados (b).

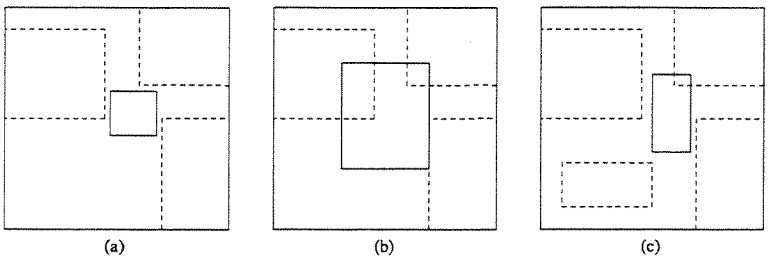



Figura 2.30: Situações especiais que podem ocorrer na inserção de um retângulo. Os retângulos tracejados representam os MBRs da entrada de um nó.

### Inserção

Para se inserir um objeto na $R^+$-tree, percorre-se a estrutura a partir da raiz e todos os nós não-folha cujos MBRs interceptem o MBR do objeto têm suas árvores percorridas. Ao atingir o nível das folhas, o retângulo é então armazenado em todas as folhas atingidas. Entretanto, existem três casos de inserção não triviais, e que não foram mencionadas em [SRF87]. O primeiro, mostrado na Figura 2.30 (a) é quando um MBR é inserido em um nó onde ele não intercepta o MBR de nenhuma das entradas. O segundo caso é quando o MBR do novo objeto intercepta apenas parcialmente algumas das entradas, como na Figura 2.30 (b). E finalmente, o terceiro é quando não há como expandir os MBRs das entradas do nó para conterem o novo retângulo sem que eles se interceptem, como na Figura 2.30 (b).

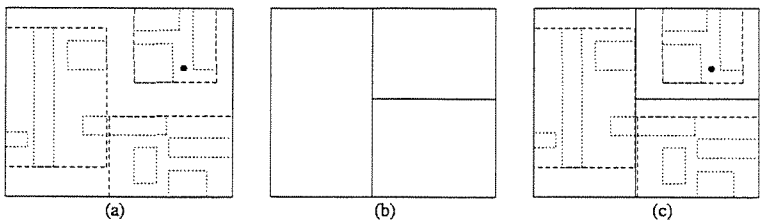



Figura 2.31: Um conjunto de entradas e seus MBRs (a), e a divisão do espaço em *MaxRects* (b) e (c).

Para resolver estes problemas, Cox, em [CM92], adotou em sua implementação uma solução que divide entre as entradas dos nós intermediários de cada nível o espaço total onde os dados estão dispostos. Assim, em qualquer nível não há espaços vagos, o que previne as situações citadas. Para isso, um novo retângulo, chamado *MaxRect*, é acrescentado a cada entrada. Esse retângulo descreve o espaço sobre o qual cada entrada é responsável, e tem as seguintes propriedades:

- todos os *MaxRects* das entradas de um determinado nó intermediário dividem completamente e sem sobreposição o espaço representado pelo *MaxRect* da entrada correspondente a esse nó em seu pai;
- caso o nó seja a raiz, a união dos *MaxRects* de suas entradas representam o espaço total onde os dados estão dispostos.

Na implementação de Cox, os *MaxRects* são utilizados para direcionar o percurso de inserção enquanto que os MBRs são utilizados para direcionar consultas. A Figura 2.31 representa um conjunto de entradas e seus *MaxRects*.

Quando ocorre um *overflow*, é necessário executar a operação de *split*, produzindo dois novos nós. O requisito de que dois nós do mesmo nível devem cobrir áreas disjunta mutuamente é satisfeito através de uma partição paralela a um dos eixos das dimensões, que irá decompor o espaço em duas sub-regiões. Como na R-tree, o *split* pode se propagar para níveis superiores, mas pode também propagar-se para níveis inferiores da árvore. Isso ocorre se a linha sobre a qual serão particionados os retângulos de um determinado nível interceptar algum retângulo nos níveis inferiores. A Figura 2.32 exemplifica essa situação: o retângulo $A$ é pai de $B$, que por sua vez é pai de $C$. Se $A$ é particionado, dando origem a $A_1$ e $A_2$, $B$ e $C$ também têm de ser divididos, uma vez que estariam representados em $A_1$ ou $A_2$ exclusivamente, o que contraria a propriedade de que a sub-árvore de uma entrada só contém um retângulo se seu MBR sobrepõe completamente este retângulo.

O algoritmo de *split* classifica os retângulos das entradas em cada eixo e utiliza um parâmetro chamado *ff (fill factor)*, que indica o número de retângulos a serem armazenados

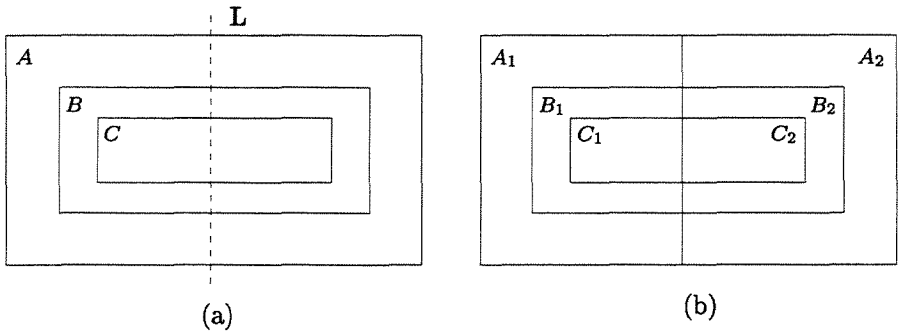



Figura 2.32: Um nó de $R^+$-tree com seus filhos (a) e os novos nós com o eixo de partição $L$ (b).

no primeiro nó resultante. Este número pode ser a capacidade de um nó ou uma fração predefinida em relação a algum fator de carga. A seleção do eixo de partição é baseado em um ou mais critérios dos especificados a seguir:

- vizinhos mais próximos;
- comprimento total dos eixos paralelos aos eixos das dimensões;
- área total mínima das sub-regiões;
- número mínimo de partições de retângulos.

Os três primeiros critérios reduzem o espaço de busca ao reduzir o *dead space*, enquanto que a minimização de *splits* do quarto item busca reduzir a altura da $R^+$-tree. Os critérios são utilizados em cada passo para encontrar uma partição no espaço que agrupe os retângulos de forma que melhore a busca.

A $R^+$-tree sofre do mesmo problema que outros métodos de acesso que utilizam a técnica do espaço sem sobreposição: ela não suporta conjuntos de dados cujas densidades sejam maiores que a capacidade de armazenamento dos nós. Isto, entretanto pode ser contornado usando um encadeamento de nós para conter os retângulos que se sobrepõem em uma área muito densa, mas a árvore ficaria desbalanceada.

### Consulta

O algoritmo de busca é similar ao utilizado na R-tree, sendo que a maior diferença entre a R-tree e a $R^+$-tree é que na primeira as sub-regiões podem se interceptar, levando a buscas mais caras. A idéia é decompor o espaço de busca em sub-regiões disjuntas e percorrer cada uma até que os objetos sejam encontrados nas folhas. Deve-se, porém, tomar o cuidado de eliminar registros repetidos do conjunto resposta, uma vez que um objeto pode estar representado em mais de uma folha.

### Remoção

Para efetuar a remoção de uma entrada de uma $R^+$-tree, deve-se removê-la de todas as folhas onde está armazenada. A identificação destas folhas pode ser feita através do procedimento de consulta. Uma vez removida a entrada, se necessário os MBRs dos nós superiores são ajustados. Uma vez que não há exigência de uma ocupação mínima, o *underflow* só ocorre quando um nó fica vazio. O *underflow* se propaga ao níveis imediatamente superiores se os nós desses níveis também tiverem somente um filho.

## 2.7.3 R*-tree

A R*-tree, proposta por Beckmann *et al* [BKSS90], procura melhorar o agrupamento de retângulos feito pela R-tree. Possui a mesma estrutura da R-tree, e os mesmos algoritmos de consulta e exclusão. A principal diferença entre elas está no algoritmo de inserção. Enquanto que na R-tree a heurística de otimização baseia-se apenas na área do retângulo, a R*-tree baseia-se em vários fatores:

- a *sobreposição* entre retângulos dos nós não-folha. A diminuição da sobreposição melhora o desempenho, uma vez que diminui o número de caminhos possíveis a serem percorridos nas consultas;
- a *ocupação* dos nós. O aumento da ocupação diminui o custo das consultas, pois reduz a altura da árvore;
- o *perímetro* dos retângulos. Assumindo uma área fixa, o objeto com menor perímetro é um quadrado. Procura-se então minimizar o perímetro ao invés da área, buscando gerar MBRs mais próximos de quadrados. Uma vez que quadrados tendem a ser agrupados mais facilmente, o agrupamento de MBRs de determinado nível resultaria em MBRs menores nos níveis superiores.

Os fatores citados acima estão concentrados na rotina que escolhe a folha onde um retângulo deve ser armazenado e na que trata do *overflow* na inserção de uma entrada.

### Percurso na Inserção

A folha onde a nova entrada será armazenada é escolhida percorrendo-se a árvore a partir da raiz. Se as entradas apontam para nós não-folha, o nó que necessitar o menor aumento de área para incluir a nova entrada é o escolhido. Empates são resolvidos escolhendo-se aquele que tiver menor área. Até esse ponto, o algoritmo é igual ao da R-tree. Se as entradas do nó escolhido apontam para folhas, a folha escolhida será aquela cujo MBR necessitar de menor aumento na somatória de suas áreas de sobreposição com os MBRs

das outras entradas do nó. Os empates são resolvidos escolhendo a entrada com menor aumento de área.

### Tratamento de *overflow*

Tanto a R-tree quanto a R*-tree são não-determinísticas ao atribuir uma entrada a um nó, ou seja, para um mesmo conjunto de dados, ordens diferentes de inserção geram estruturas diferentes. Retângulos inseridos no início da construção podem gerar MBRs nos níveis superiores que levarão a uma degradação no desempenho à medida que novas entradas são inseridas.

Uma vez que durante o *split* apenas uma reorganização local é feita, Beckmann *et al* propõem a reinserção de parte das entradas como forma de melhorar o desempenho da estrutura. Este procedimento é semelhante ao utilizado na R-tree para o tratamento de *underflow* após a exclusão de uma entrada.

O algoritmo de reinserção classifica em ordem decrescente as entradas de um nó $N$ utilizando a distância entre o centro do retângulo da entrada e o centro do retângulo de $N$. Após a classificação, $p$ entradas são removidas, enquanto que as entradas restantes continuam em $N$, que tem seu MBR ajustado para conter as novas entradas. Feito isso, as $p$ entradas são reinseridas, podendo-se utilizar dois critérios para reinserção: do mais distante para o menos distante do centro de $n$ (*far reinsert*), ou do menos distante para o mais distante (*close reinsert*). Nos experimentos mostrados em [BKSS90], os melhores desempenhos foram obtidos utilizando $p$ igual a 30% de $M$, onde $M$ é o número máximo de entradas permitidas em um nó.

É possível que as entradas venham a ser inseridas no mesmo nó em que estavam, podendo sofrer novo *overflow*. Da mesma forma, pode ocorrer que um ou mais nós do mesmo nível, recebendo as entradas, sofram *overflow*. Para evitar que a rotina entre em *loop*, a reinserção é utilizada apenas na primeira vez que ocorre *overflow* em determinado nível. A partir da segunda vez é utilizada a rotina de *split*. Segundo Beckmann *et al*, o custo maior de CPU na utilização da reinserção é compensado pela menor necessidade de se efetuar operações de *split*.

O algoritmo de *split* da R*-tree inicialmente classifica as entradas do nó, em cada eixo de dimensão, primeiro pelo valor mínimo e depois pelo valor máximo de coordenada do MBR da entrada. Para cada classificação, são determinadas $M - 2m + 2$ distribuições das $M + 1$ entradas em dois grupos, onde a $k$-ésima distribuição ($1 \leq k \leq M - 2m + 2$) é composta por $(m - 1) + k$ entradas atribuídas ao primeiro grupo e as restantes atribuídas ao segundo grupo. Para cada distribuição calcula-se:

- a área da distribuição, que é a soma das áreas dos dois retângulos que envolvem os grupos;

- o perímetro da distribuição, que é a soma dos perímetros dos dois retângulos que envolvem os grupos;
- a área de interseção dos dois retângulos que envolvem os grupos.

A dimensão onde será feita a partição é a que apresentar menor somatória dos perímetros de todas as distribuições. Escolhida a dimensão, deve-se escolher a distribuição com menor área de interseção, sendo que empates são resolvidos escolhendo-se a distribuição com menor área.

### 2.7.4 Hilbert R-tree

Proposta por Kamel e Faloutsos [KF94], a Hilbert R-tree é um método de acesso baseado na R-tree que procura adiar ao máximo a execução das rotinas de *split*. A idéia sobre a qual se baseia a Hilbert R-tree é criar uma vizinhança $s$ de nós, utilizando-se para isso uma ordenação de nós através da curva de Hilbert, uma *space-filling curve* (Seção 2.4.1). Assim, se um nó tiver sua capacidade ultrapassada, ao invés de se executar o *split*, procura-se passar algumas de suas entradas aos seus $s-1$ vizinhos, adiando o *split* ao máximo. O *split* só é executado caso todos os nós de $s$ estiverem cheios. Esta característica da Hilbert R-tree permite que, ajustando-se $s$,a ocupação dos nós fique muito próxima de 100%.

A Hilbert R-tree possui a seguinte estrutura:

- nós folha, contendo no máximo $C_f$ entradas do tipo *(identificador, MBR)*, onde $C_f$ é a capacidade da folha, identificador é um ponteiro para a descrição do objeto e MBR é seu retângulo envolvente mínimo;
- nós não-folha, contendo no máximo $C_n$ entradas do tipo *(ponteiro, MBR, LHV)*, onde $C_n$ é a capacidade do nó, ponteiro indica um nó filho, MBR é o retângulo envolvente mínimo da sub-árvore apontada por *ponteiro* e LHV é o maior valor na curva de Hilbert entre os retângulos de dados[3] envolvidos por MBR. Os valores LHV são calculados tomando-se os centros dos MBRs.

#### Inserção

Para se inserir um retângulo $r$ na Hilbert R-tree, o valor de Hilbert $h$ do centro de $r$ é utilizado como chave. Percorre-se a árvore a partir da raiz e, em cada nível escolhe-se o

[3]As coordenadas dos MBRs dos nós não-folha nunca são calculadas.

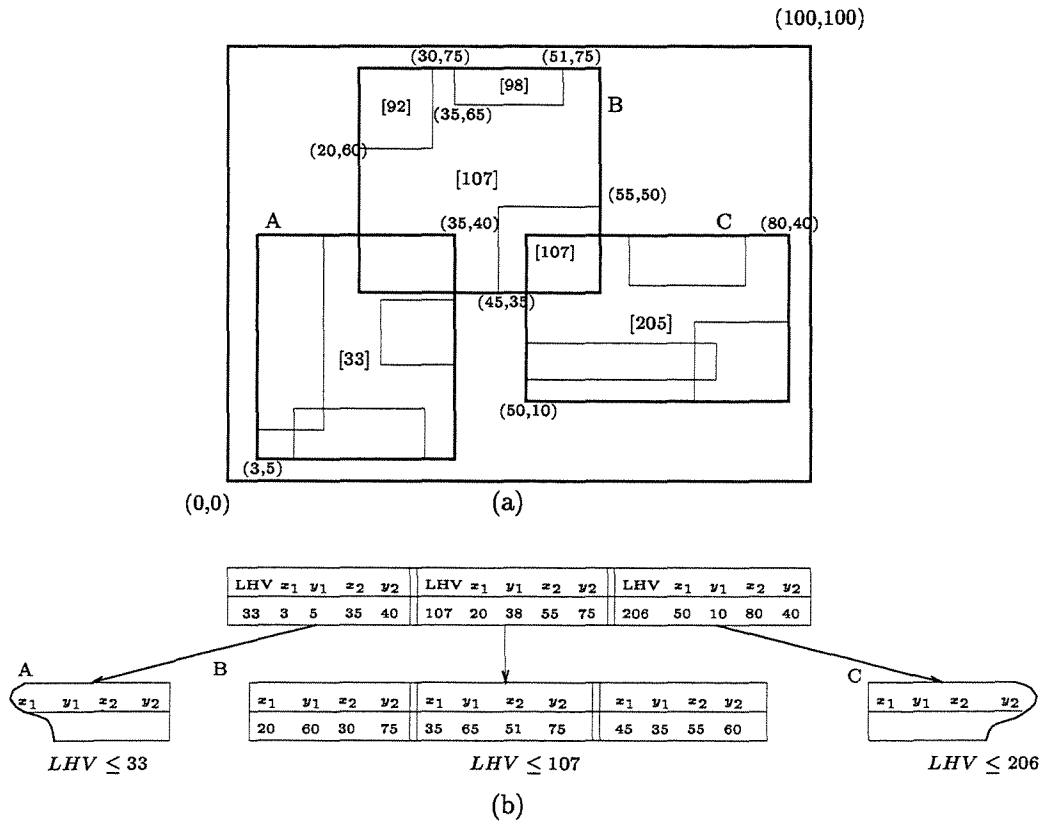



Figura 2.33: Os retângulos indexados de uma Hilbert R-tree (a). Entre colchetes estão as coordenadas da curva de Hilbert enquanto que entre parênteses estão as coordenadas no espaço. A árvove é apresentada na Figura (b).

nó cujo $LHV$ seja o menor dentre os maiores ou iguais a $h$, até atingir uma folha $L$. Caso $L$ tenha menos de $C_f$ entradas, $r$ é inserido e são feitos ajustes nos nós superiores.

Caso tenha ocorrido *overflow* em $L$, seja $\varepsilon$ o conjunto de todas as entradas de uma vizinhança $s$, ou seja, de $L$ e seus $s-1$ vizinhos. Adicionamos $r$ a $\varepsilon$. Assim, se pelo menos um dos $s-1$ vizinhos de $L$ não estiver cheio, distribui-se $\varepsilon$ entre os nós de $s$ de acordo com o valor de Hilbert. Caso contrário, cria-se um novo nó $LL$ e distribuem-se os elementos de $\varepsilon$ entre os nós de $s+1$.

Como na R-tree, o *split* pode se propagar até os níveis superiores da árvore. Mas ocorra ou não o *split*, deve-se ajustar os MBRs e os LHVs das entradas do caminho até a folha na qual $r$ foi inserido.

A ocupação na árvore cresce à medida que o número de elementos em $s$ cresce. Entretanto, o custo da inserção também aumenta em relação ao número de nós lidos. Isto é devido ao fato de que a cada nó cheio de $s$ encontrado, é necessário ler seus $s-1$ vizinhos para que seja feita a redistribuição das entradas. Kamel e Faloutsos, em [KF94] citam a política de *split 2-para-3* como a que possui melhor compromisso entre desempenho e custo de inserção.

#### Consulta

As rotinas de consulta são exatamente iguais as da R-tree.

#### Remoção

Na Hilbert R-tree, ao invés de se reinserir entradas em caso de *overflow*, como na R-tree ou R*-tree, o que se procura fazer é emprestar algumas entradas de seus $s$ vizinhos[4]. Se todos os vizinhos já estiverem com ocupação minima, o nó em questão e seus $s$ vizinhos são condensados em $s$ nós. O *underflow* pode se propagar até os níveis superiores da estrutura.

Como na inclusão, os nós pertencentes ao caminho percorrido da raiz até à folha que teve entradas removidas têm seus MBRs e LHVs ajustados, se necessário.

### 2.7.5 R-tree compactas

A R-tree e suas variantes foram projetadas para ambientes dinâmicos, onde teriam de suportar inserções e remoções alternadas e freqüêntes de forma eficiente. Estas estruturas garantem que a utilização do espaço seja de pelo menos 50%. Em algumas experiências [BKSS90] mostrou-se que a utilização média é de 68%, aproximadamente. Entretanto, para um conjunto estático de dados, uma melhora na utilização do espaço traria dois

[4]Note que são necessários $s-1$ vizinhos para se inserir e $s$ para remover.

grandes benefícios: reduz o espaço ocioso ocupado pela estrutura além de oferecer melhor tempo de resposta, uma vez que a estrutura terá *fanout* maior e possivelmente a árvore terá altura menor.

Dados estáticos aparecem em aplicações como: bases de dados cartográficas, onde inserções e remoções são raras; bancos de dados com dados censitários; bancos de dados publicados em CD-ROMs; dados meteorológicos, temporais ou ambientais. Entretanto, embora raramente sofram modificações estas bases de dados são muito grandes, e portanto, qualquer diminuição no espaço ocupado tende a ser crucial.

Os três métodos de compactação [KF93, RL85, LLE97] utilizam um processo similar para a construção do índice. Em um espaço com $r$ retângulos, considerando que cada nó comporta $b$ retângulos, os passos para se criar uma estrutura compacta são os seguintes:

1. o arquivo de dados é processado e os $r$ retângulos são ordenados em $\lceil r/b \rceil$ grupos consecutivos de $b$ retângulos onde cada grupo de $b$ deve ser colocado no mesmo nó do nível das folhas. O último grupo pode ter menos de $b$ retângulos;

2. os $\lceil r/b \rceil$ são então carregados em páginas e os pares (*MBR*, *número da página*) para cada folha são armazenados em um arquivo separado. O campo *número da página* será utilizado como ponteiro para o nó do nível imediatamente superior;

3. os pares (*MBR*, *número da página*) são então armazenados em nós no próximo nível, recursivamente, até a raiz.

Estes métodos diferem apenas no critério de ordenação dos retângulos quando os mesmos são processados. A primeira proposta é o *Nearest X* [RL85]. Por este método, os retângulos são classificados utilizando-se a coordenada $x$ e armazenados em nós com capacidade para armazenar $b$ retângulos. O segundo método é o *Hilbert Sort* [KF93]; esta proposta classifica os retângulos utilizando o valor na curva de Hilbert dos pontos centrais de cada MBR, e então procedendo como no Nearest X.

A última proposta é o *Sort-Tile-Recursive* [LLE97], cuja idéia é dividir o espaço utilizando $\sqrt{r/b}$ "fatias" de forma que cada fatia tenha retângulos suficientes para preencher $\sqrt{r/b}$ nós. Para um espaço bidimensional, fazendo $P = \lceil r/b \rceil$ número de folhas, os retângulos são classificados pela coordenada $x$ do centro do MBR e particionados em $\sqrt{P}$ "fatias" verticais. Em cada partição, os retângulos são classificados em relação à coordenada $y$ do centro do MBR e são então armazenados em nós. Note que o último grupo pode ter menos de $b$ retângulos. Os níveis superiores são construídos recursivamente utilizando o mesmo método descrito. Para dimensionalidades maiores, os hiper-retângulos são classificados.

Este método pode ser generalizado para espaços onde a dimensionalidade $k > 2$: inicialmente os hiper-retângulos são classificados em relação à primeira coordenada de

seus centros. Então divide-se o conjunto de entrada em $S = \lceil P^{\frac{1}{k}} \rceil$ fatias, onde cada fatia corresponde a um conjunto de $b \times \lceil P^{\frac{k-1}{k}} \rceil$ hiper-retângulos consecutivos de uma lista. Cada fatia é então processada recursivamente utilizando as $k-1$ coordenadas restantes.

### 2.7.6 Outras pesquisas sobre R-tree e variantes

As R-trees tornaram-se uma classe de índices amplamente utilizada em sistemas comerciais de bancos de dados devido à sua relativa simplicidade de implementação e desempenho muito bons nas operações de consulta, inserção e remoção. Desta forma, é natural que as R-tree e suas variantes se tornem alvo de pesquisas como paralelização e operações intensivas, dentre outras. Nesta seção, veremos algumas pesquisas que têm sido desenvolvidas de forma a melhorar o desempenho da família das R-tree em suas diversas aplicações.

Em [TS93], Theodoridis e Sellis propõem novos algoritmos de inserção e de *split* para melhorar o desempenho das R-tree convencionais, levando em consideração fatores como a área de sobreposição, a área de cobertura e o *dead space* dos nós. Os resultados obtidos mostraram que os fatores considerados aumentaram o desempenho da R-tree aos níveis da R*-tree.

A paralelização de R-trees é investigada em [KF92] por Kamel e Faloutsos e em [SL99] por Schnitzer e Leutenegger. Estes trabalhos propõem e avaliam vários métodos de distribuição das páginas de dados entre os discos, bem como métodos de agrupamento dos dados nas páginas. Schnitzer e Leutenegger ainda investigam diferentes configurações de dados (sintéticos e reais) e diferentes tamanhos de consultas.

Duas categorias de *bulk loading* foram propostas: a primeira aplica uma construção *bottom-up* (veja a seção 2.7.5). A outra classe concentra-se em uma abordagem *top-down*. Em [BSW97] Bercken, Widmayer e Seeger propõem a carga da R-tree utilizando uma estrutura baseada em memória chamada *buffer tree* [Arg95]. García, Lópes e Leutenegger [GLL97] propõem uma técnica que particiona os dados de forma que minimiza uma função (critério) definida pelo usuário. Este algoritmo produz uma árvore ótima, mas demanda muito tempo para a carga. Em [CCR98], Chen, Coubey e Rundensteiner propõem uma técnina denominada STLT (*Small Tree Large Tree*). Esta técnica, dadas as entradas, constói uma árvore, chamada *Small Tree* de altura $h$ e então procura um local apropriado na árvore original *Large Tree* de altura $H$, no nível $L = H - h$.

Em [AHVV99] Arge *et alii*, propõem o uso da *buffer tree*, citada anteriormente, na execução de operações intensivas em R-tree dinâmicas.

Alguns trabalhos também têm sido feitos na área de modelos analíticos de desempenho de R-trees. Modelos analíticos são úteis porque: permitem melhor compreensão do comportamento da estrutura de dados sob conjuntos de dados com características variadas; podem ser utilizados quando várias propostas de indexação precisam ser comparadas,

sem que as propostas tenham sido previamente implementadas; e, podem ser utilizadas por otimizadores de consultas para avaliar os custos de uma consulta espacial complexa e seu procedimento de execução. Em [TS95, TSS99], são apresentados modelos analíticos para previsão de desempenho, em número de acessos a disco, de R-trees na execução de consultas, baseando-se nas características dos dados como densidade, quantidade e outras.

# Capítulo 3

# Métodos de Junção Espacial – classificação e propostas

## 3.1 Introdução

Junções espaciais são uma classe importante de operações espaciais e, devido a este fator, têm sido alvo de intensas pesquisas. São operações utilizadas para se responder perguntas do tipo: "Quais são os prédios públicos que estão próximos a praças?" ou "Quais estradas cruzam rios?".

Pode-se definir junção, de um modo formal, como um subconjunto do produto cartesiano de dois conjuntos $A$ e $B$, não necessariamente distintos, com cardinalidade $|A|$ e $|B|$, respectivamente. O subconjunto resultante é formado pelos pares de elementos de $A$ e $B$ que satisfazem um predicado $p$.

As junções espaciais são análogas às junções naturais, que envolvem atributos escalares e, como tais, exigem grande esforço de computação. Entretanto, nas junções espaciais, ao invés de dados convencionais, dados espaciais são utilizados como critério de junção na satisfação de predicados de junção. Dentre estes predicados, destacamos "interseção", "contém", "em direção a" e "à distância de", dentre outros. Neste fato reside outra diferença entre junções espaciais e junções convencionais: enquanto que na última utiliza-se, mais comumente, o critério da igualdade, na primeira o predicado mais comum é a interseção entre objetos. Os predicados utilizados em operações de junção espacial dificultam enormemente a utilização de algoritmos aplicados em junções convencionais, intensivamente já estudados.

Este capítulo procura fazer uma compilação dos esforços na área de junções espaciais, embora não tenha a pretensão de ser uma compilação completa, pois devido às diversas pesquisas efetuadas na área, tal tarefa seria extremamente difícil. Para este fim, este capítulo está organizado da seguinte forma: a seção 3.2 traz uma visão geral de junções

sobre dados convencionais; a seção 3.3 traz alguns conceitos sobre junções espaciais, suas subdivisões e os métodos propostos em cada subdivisão, além de outras pesquisas envolvendo junções espaciais.

## 3.2 Junções sobre dados convencionais

Junções convencionais (relacionais) são aquelas executadas sobre dados escalares, ou seja, dados alfanuméricos. São operações comumente encontradas em sistemas gerenciadores de bancos de dados e executadas sobre dados como nomes, endereços, valores, dentre outros.

Em junções convencionais, geralmente, procuram-se tuplas que possuam valores iguais em um determinado atributo, embora outros predicados como $<$, $>$, $=$, dentre outros, também possam ser utilizados. Este tipo de operação já foi intensivamente estudada, podendo ser realizada de três formas distintas: *nested loops*, *sort-merge* e *hash-join*.

A abordagem de *nested loop* se resume em avaliar o predicado da junção a todos os pares resultantes da operação $A \times B$. Esta abordagem possui desempenho extremamente fraco e deve ser considerada apenas em casos extremos de implementação. Entretanto, esta abordagem satisfaz qualquer predicado, uma vez que a junção é considerada como uma restrição sobre o produto cartesiano.

Se as duas relações podem ser classificadas em relação aos atributos $i$, $i \in A$ e $j$, $j \in B$, que serão considerados na junção, há maneiras mais eficientes de se computar a mesma. A estratégia de *sort-merge* primeiramente classifica $A$ em relação à coluna $i$ e depois classifica $B$ em relação à coluna $j$, restringindo o espaço de busca. Percorre-se então as duas relações, juntando os elementos de $A$ com os elementos de $B$. Este algoritmo pode ser executado em tempo proporcional a $\mid A \mid log \mid A \mid + \mid B \mid log \mid B \mid + \mid J \mid$, onde $\mid J \mid$ é a cardinalidade do resultado da junção. Note que no pior caso $\mid J \mid = \mid A \mid \cdot \mid B \mid$.

O *hash-join* é uma técnica que possui desempenho excelente, particularmente para relações grandes comparadas ao tamanho do *buffer*. Esta técnica divide-se em duas fases: fase de partição e fase de junção. Durante a fase de partição, os conjuntos internos e externos são particionados, atribuindo-os a seus respectivos *buckets* através de uma função de partição (funções de *hash*). A fase de junção é a responsável pela verificação dos *buckets*, aplicando o predicado em cada um dos elementos, gerando a resposta da operação de junção.

Entretanto, uma abordagem diferente pode ser tomada se a freqüência de atualização nos bancos de dados for baixa. Pode-se pré-computar as junções mais freqüentes, e armazenar os resultados em uma estrutura denominada *join index* [Val87]. Um *join index* é uma relação de duas colunas, contendo os identificadores das tuplas que se combinam, satisfazendo determinado predicado. Ao se realizar uma junção, consulta-se o *join index*

correspondente e recuperam-se as tuplas desejadas. Se considerarmos o tempo de computação de uma junção, esta estratégia é altamente eficaz. Entretanto, dependendo da seletividade da junção, os requisitos de armazenamento do *join index* podem ser muito altos.

Duas características de junções espaciais impedem a aplicação direta de algoritmos de junção relacional a junções espaciais. Primeiramente, dados espaciais não possuem uma ordenação natural total que preserve a proximidade espacial. Embora existam algoritmos baseados em *space-filling curves* (veja seção 2.4.1), que definem uma ordenação sobre os conjuntos de objetos espaciais, esta ordenação não preserva proximidade espacial uniformemente. Desta forma, dois objetos espacialmente próximos podem estar distantes na ordenação. A segunda dificuldade é que junções relacionais são otimizadas para realização de junções de igualdade. Entretanto, os predicados analisados em junções espaciais são, frequentemente, muito mais complexos.

Desta forma, torna-se necessário o desenvolvimento de algoritmos apropriados ao domínio espacial, que ofereçam bom desempenho quanto à avaliação de predicados e em relação a requisitos de I/O. Muitas foram as respostas a esses requisitos, como veremos na seção seguinte.

## 3.3 Junções espaciais

Uma junção espacial entre as relações $A$ e $B$ é uma operação que constrói pares de tuplas a partir de $A \times B$, cujo atributo espacial satisfaz um predicado espacial. Há vários predicados espaciais, dentre os quais podemos citar "interseção", "contém", "à distância de", "em direção a", sendo que o predicado mais comum é interseção. Devido a este fator, este trabalho se restringe às operações de junção de interseção.

Geralmente, junções espaciais e outras consultas espaciais são executadas em dois passos:

- filtragem, onde as tuplas cujos MBRs satisfazem o predicado são selecionadas;
- e refinamento, onde a forma dos objetos é recuperada e o predicado espacial é verificado com relação à esta forma.

Alguns autores [BKSS94, BK94] propõem ainda uma fase intermediária entre as fases de filtragem e de refinamento, denominada fase de *aproximação*. Esta fase se daria através da utilização de outras abstrações além do MBR. Os autores sugerem a utilização do polígono envolvente mínimo de 5 vértices (5-C) na identificação de *false hits* e a utilização de aproximações conservativas na identificação de *hits*. Um objeto teria sua geometria

recuperada apenas se não fosse possível identificá-lo através de suas outras aproximações (veja seção 2.2 para maiores detalhes).

Vimos na seção 2.2 que, se dois objetos se inteceptam, então seus MBRs também se interceptam, embora a recíproca não seja verdadeira. Isto também vale para outros predicados, como exemplo "contém", "toca", dentre outros. Isto se deve ao espaço ocupado pela abstração espacial do objeto e que não é ocupado pelo mesmo, o chamado *dead space.*

Para evitar que objetos que satisfazem o predicado sejam descartados na fase de filtragem, predicados mais complexos são substituídos por predicados mais simples. Assim, na fase de filtragem, um predicado $a\theta b$, mais específico, é substituído por outro predicado $a\Theta b$ mais geral. Como exemplo, se dois objetos se "tocam", é provável que seus MBRs se interceptem ao invés de simplesmente se "tocarem", e esses objetos seriam descartados. A substituição do predicado "toca" por "intercepta" faz com que todos os objetos que se tocam sejam recuperados, muito embora objetos que não se tocam também o sejam. Entretanto não perderíamos aqueles que realmente satisfazem o predicado.

A grande variedade de métodos propostos e a própria complexidade da operação de junção torna difícil uma comparação justa entre diferentes métodos. Mesmo assim, várias tentativas têm sido feitas. A seguir, apresentaremos e discutiremos alguns métodos de junção propostos, bem como algumas tentativas de se compará-los, e outras pesquisas sobre junções espaciais.

### 3.3.1 Classificação de métodos de junções espaciais

Os algoritmos de junção espacial, dependendo se as relações são indexadas ou não, podem ser classificados em três categorias [PRS99], a saber:

- sem índices;
- apenas uma relação indexada:
- ambas indexadas:

Quando não há índices disponíveis sobre determinado conjunto, é provável que este conjunto seja o resultado de uma consulta feita previamente. Pode-se então aplicar técnicas de particionamento, que separam as tuplas em *buckets* e então utilizam alguma técnica de *plane-sweeping* ou *hashing* sobre os *buckets.*

Se uma das relações está indexada e a outra não, pode-se optar por uma abordagem *scan-and-index* ou construir um índice em tempo de execução e executar a junção.

Finalmente, se os dois conjuntos estão indexados, um método geral de junção pode ser aplicado, especializando-o ao de método de acesso utilizado. Um método de grade pode

utilizar alguma técnica de *hashing* e um método baseado em árvore pode utilizar uma técnica de percurso em árvores.

Nas subseções seguintes veremos mais detalhadamente propostas que se incluem em cada uma dessas subdivisões. Com relação aos métodos de junção sobre duas relações indexadas, nos restringiremos ao caso onde o método de acesso empregado é uma estrutura em árvore, com ênfase nas R-trees, foco de nosso trabalho.

### Métodos aplicados sobre duas relações indexadas

A primeira proposta de métodos de percurso sincronizado em árvores (*synchronized tree traversal* – STT) foi feita por Günther em [Gün93]. Este método seria executado sobre um tipo de estrutura denominada *generalization trees* (GTs), uma estrutura em árvore onde cada nó corresponde a um objeto espacial e que possuem as seguintes características:

- exceto pela raiz, cada nó está completamente contido no objeto correspondente a seu nó pai;
- objetos diferentes podem se sobrepor;
- objetos de quaisquer níveis não necessariamente têm que cobrir o espaço total, ou seja, permite-se *dead space.*

A definição geral de *generalization trees* inclui a R-tree e variantes (veja seção 2.7), dentre outras estruturas.

A partir da definição de GTs, o autor define um algoritmo *depth-first* de junção espacial baseado em GTs, efetuando um estudo comparativo entre os algoritmos *nested loop* e o método de *join index*, como descrito em [Val87], sobre dados sintéticos. O autor define também modelos analíticos de custo para as operações de atualização, seleção e junção espacial para os três métodos estudados (veja seção 3.3.2).

Os experimentos de Günther mostram que o desempenho de uma junção espacial depende da seletividade da junção. Para junções com baixa seletividade, a melhor escolha são os *join indices.* Para junções com alta seletividade, as GTs aliadas a algoritmos STT são a melhor escolha.

Outro trabalho que propõe algoritmos STT é o proposto em [BKS93] por Brinkhoff, Krigel e Seeger. Os autores também propõem a utilização de uma abordagem *depth-first* muito parecida à proposta por Günther, mas incorporam importantes melhoramentos ao algoritmo. Os algoritmos propostos por Brinkhoff *et alii* incluem otimizações para execução sobre R-trees e que podem também ser aplicadas sobre outras estruturas.

O algoritmo básico proposto por Günther e Brinkhoff, para árvores de mesma altura, é o seguinte:

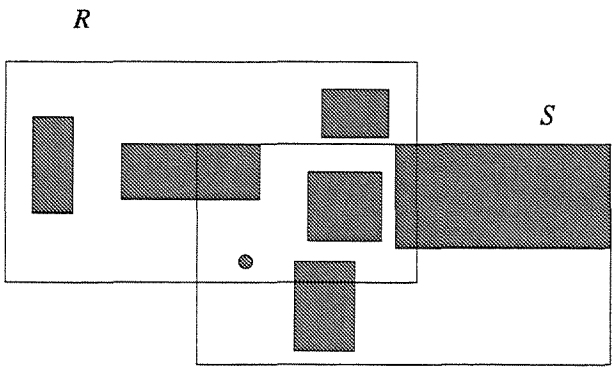



Figura 3.1: Restrição do espaço de busca em uma R-tree.

```
Junção_em_profundidade(R, S: nó)
   Para (todo r de R) Faça
      Para (todo s de S tal que r.MBR intercepte s.MBR) Faça
         Se (R for página) Então
            Inclua (r, s) no conjunto-resposta;
         Senão
            Junção_em_profundidade(r.ponteiro, s.pointeiro);
 FimSe
      FimPara
   FimPara
Fim
```

Quando as árvores possuem alturas diferentes, deve-se executar uma *intersection query* utilizando como janela de consulta os MBRs das entradas do nó folha sobre os nós da árvore mais alta.

As otimizações propostas por Brinkhoff *et alii* incluem a restrição do espaço de busca, classificação espacial e varredura de planos (*plane-sweeeping*) e fixação de páginas de disco no *buffer* (*buffer pinning*).

A restrição do espaço de busca consiste em verificar apenas as entradas que se encontram na área de interseção dos nós. Isto pode ser melhor visualizado na Figura 3.1. Com isso, poupa-se o tempo de verificar todas as entradas de $R$ em relação a todas as entradas de $S$, bastando verificar apenas os MBRs que interceptam a área de interseção, uma vez que apenas esses têm a possibilidade de se interceptarem.

Outro melhoramento é a introdução de técnicas de *plane-sweeping* aliada a uma classificação das entradas para melhorar o desempenho do algoritmo em relação à verificação de interseção entre os MBRs.

Estas modificações visam reduzir o impacto do tempo de CPU na execução. Para reduzir o impacto do tempo de I/O, os autores propõem uma *política de otimização local baseada em localidade espacial.* Esta política nada mais é do que uma política de seqüenciamento de leitura das páginas do disco. Os autores investigaram três políticas: organização gerada pela operação de *plane-sweeping*; organização gerada pela operação de *plane-sweeping* com fixação da página mais utilizada no *buffer* (*buffer pinning*); organização através de uma curva $z$.

A proposta mais recente de métodos STT é apresentada por Huang *et alii* [HJR97]. Neste trabalho, os autores propõem um percurso *breadth-first* para a realização da junção. Entretanto, a proposta ainda inclui os vários melhoramentos introduzidos por Brinkhoff *et alii*, como *plane-sweeping* e *buffer pinning.* O algoritmo básico para a execução sobre árvores de mesma altura é o seguinte:

```
Junção_em_largura(R, S: nó)
// hR = altura da árvore com raiz R; hS = altura da árvore com raiz S
// Lista_junção[i] é o índice de junção intermediário criado no nível i
// junção(p, q) é uma função que executa junção de dois nós e devolve
// uma lista contendo pares <r.ponteiro, s.ponteiro> de elementos que
// se interceptam

Var Lista_junção[hR] := ∅;

   Lista_junção[i] := junção(R, S);
   Enquanto i < hR Faça
      Para (todo elemento <r, s> de Lista_junçao[i]) Faça
         Lista_junção[i + 1] := junção(r.ponteiro, s.ponteiro);
      FimPara
      i := i + 1;
   FimEnquanto
Fim
```

Os autores justificam a execução em largura através da inserção de *otimizações globais.* Na junção em profunidade, as otimizações são *locais.* Isto significa que, enquanto a otimização no último método aplica-se apenas ao resultado da junção de dois nós em cada nível visitado, as otimizações do primeiro método aplicam-se a todos os elementos de determinado nível, uma vez que se executa a junção em todos os nós do nível $i$ antes de avançar para o nível $i + 1$. Além de incluir todos os melhoramentos apresentados por Brinkhoff [BKS93], Huang *et alii* ainda investigam o desempenho de diferentes

classificações dos índices intermediários, e o impacto do armazenamento dos índices intermediários em disco, uma vez que, ao percorrer as estruturas em largura, os índices intermediários podem atingir tamanhos proibitivos para o armazenamento em memória.

### Métodos aplicados sobre apenas uma relação indexada

A ausência de um dos índices em uma junção espacial pode ser justificada pela existência de consultas prévias de seleção ou até mesmo de junção sobre determinados conjuntos de dados. Surge então a questão: é preferível construir um índice sobre o conjunto não indexado ou a melhor saída é uma abordagem *scan-and-index*?

Uma investigação sobre esse assunto é feita por Lo e Ravishankar em [LR94]. Os autores propõem a utilização de uma estrutura denominada *seeded tree* em ambientes onde o método de indexação é baseado nas R-trees. Uma *seeded tree* (ST) é uma estrutura criada dinamicamente cujo algoritmo de construção utiliza informações sobre a operação de junção e os conjuntos de dados disponíveis, sendo aplicada na redução dos custos de junção.

A idéia da *seeded tree* é copiar os $k$ primeiros níveis da R-tree para guiar a inserção dos elementos pertencentes ao conjunto não indexado. Estruturalmente, uma ST consiste de dois níveis: o *nível das sementes* e o *nível de crescimento.*

O nível das sementes inicia-se na raiz e se prolonga até alguns níveis. É neste nível onde se encontram os primeiros níveis copiados da estrutura já existente. Ele servirá como "guia" para a inserção do conjunto não indexado. Os dados inseridos na ST irão concentrar-se no nível de crescimento.

Na Figura 3.2 apresentamos um exemplo da construção de uma ST com *fan-out* igual 2: inicialmente a ST apresenta apenas o nível das sementes, vazios, representado pelas caixas vazadas; insere-se então os retângulos $r_1$ e $r_2$ através do *slot* $S_1$ e o retângulo $r_3$ através do *slot* $S_2$, gerando os nós $G_1$ e $G_2$, respectivamente, que já fazem parte do nível de crescimento (representado pelas caixas pretas); a tentativa de inserir um novo retângulo $r_4$ em $G_1$ causa *overflow* em $G_1$, criando os nós $G_3$ e $G_4$. Assim, esta sub-árvore passará a ter como raiz o nó $G_4$.

Note que uma ST não precisa estar balanceada. Entretanto, as sub-árvores pertencentes ao nível de crescimento devem estar balanceadas. Pode-se considerar o nível de crescimento como uma floresta de R-trees.

Lo e Ravishankar investigaram vários métodos de construção do nível de sementes e do nível de crescimento, bem como experiências para o melhoramento do desempenho das estruturas. Nas investigações sobre o desempenho da estrutura na realização de junções, a ST obteve resultados muito bons em relação aos outros métodos testados. Os outros métodos avaliados foram os seguintes: utilizar os elementos da relação não indexada como janelas de consulta sobre a R-tree (*scan-and-index*); construir de uma R-tree a partir do

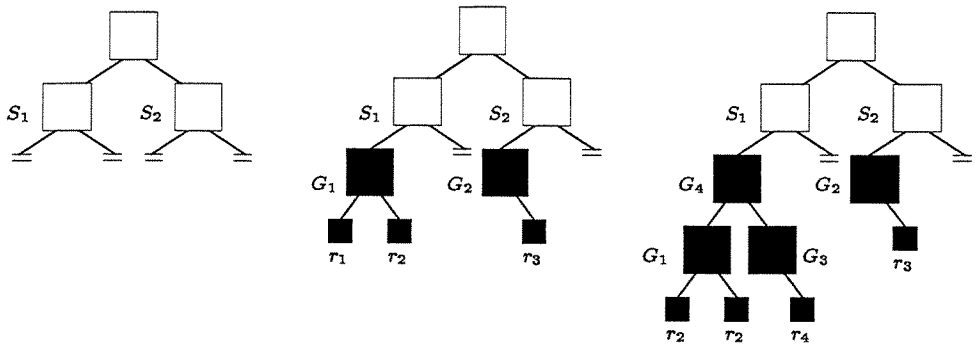



Figura 3.2: Processo de construção de uma *seeded tree*.

conjunto não indexado e a executar o algoritmo proposto em [BKS93] utilizando os dois índices como entrada (veja seção 3.3.1).

Em [MP99], Mamoulis e Papadias apresentam um novo algoritmo que executa a junção espacial entre duas entradas, uma não indexada e outra indexada através de uma R-tree. Este novo algoritmo, denominado *SISJ* (*slot index spatial join*), foi proposto em lugar da junção espacial baseada em *seeded trees*, pois os autores mostraram que as STs são muito suscetíveis a limitações do tamanho do *buffer*.

A motivação deste método é aplicar *hash-join*, utilizando os nós superiores da R-tree como *buckets*, levando a um número de partições (*slots*) $S$ menor que o tamanho do *buffer* $M$. Estes *slots* possivelmente se sobrepõem. Cada *slot* contém o MBR da entrada da R-tree correspondente, juntamente com uma lista de ponteiros para as entradas da R-tree contidas neste MBR. O algoritmo utiliza estes MBRs para executar o *hashing* do segundo conjunto (não indexado). O *hash-join* é então executado, juntando cada *bucket* do conjunto $B$ com os dados das entradas de R-trees apontados pelo *slot* correspondente.

Os autores investigaram o desempenho deste algoritmo utilizando variados métodos de particionamento, e executanto os testes sobre conjuntos de dados sintéticos e reais. Os outros métodos utilizados na comparação foram o *hash-join* de Lo e Ravishankar [LR96] (veja seção seguinte) e a junção utilizando o método de Huang *et alii* [HJR97] de busca em largura, apresentado na seção anterior.

### Métodos aplicados sobre duas relações não indexadas

A ausência de índices sobre objetos espaciais, dentre outros motivos, é resultante de consultas prévias sobre os dados ou ainda em ambientes paralelos, onde a redistribuição

de entradas é dinâmica. Isto implica que, frequentemente, haverá casos onde índices não estarão disponíveis para a realização de junção. Assim, várias pesquisas têm sido feitas para investigar a viabilidade de se construir índices em tempo de execução da junção ou até mesmo de encontrar alternativas a esta abordagem.

A primeira tentativa neste sentido é apresentada em [PD96], onde Patel e DeWitt propõem o algoritmo de junção PBSM (*Partition Based Spatial-Merge Join*), uma generalização do algoritmo *Sort-Merge* relacional. Como grande parte dos algoritmos de consulta sobre dados espaciais, este também está dividido nas fases de filtragem e de refinamento.

A fase de filtragem do PBSM inicia-se pela leitura das tuplas de um conjunto de entrada $R$. Cada tupla deste conjunto, constituída de pares $(MBR, id)$, é adicionada a uma relação temporária no disco denominada $R'$. O mesmo acontece à outra relação, $S$. Agora, deve-se encontrar os pares $\{(r,s) |\ r \in R, s \in S\}$ cujos MBRs satisfazem o predicado de junção. Se $R'$ e $S'$ cabem na memória, há vários algoritmos de *plane-sweeping* que podem ser utilizados para a computação do predicado. Assim, o resultado desta computação é passado para a próxima fase, a fase de refinamento.

Caso as duas relações não possam ser armazenadas inteiramente na memória, então cada relação é dividida em $P$ partições não disjuntas $R'_1$, $R'_2$, $\cdots$, $R'_P$ e $S'_1$, $S'_2$, $\cdots$, $S'_P$. As partições são formadas de maneira que, para cada elemento $r = (MBR, id)$ de uma partição $R'_i$, todos os elementos de $S'$ cujos MBRs interceptam o MBRs de $r$ estão presentes em $S'_i$. Além disso, o tamanho das partições é determinado de forma que para todo $i$, $1 \leq i \leq P$, as partições $R'_i$ e $S'_i$ cabem, ao mesmo tempo, na memória.

A formação destas partições é feita através de uma *função de partição*, que funciona da seguinte forma:

1. a partir de informações sobre o atributo de junção da relação $R$, o algoritmo estima o *universo*[1] do conjunto de entrada;

2. o universo é então decomposto em $P$ subpartes;

3. aplica-se a função de particionamento sobre o MBR das tuplas. A função de particionamento determina todas as subpartes do universo que determinado MBR intercepta e insere a tupla na partição correspondente. Uma vez que as partições não são disjuntas, se um MBR interceptar mais de uma subparte, as tuplas serão inseridas em todas as partições correspondentes.

Depois que os conjuntos $R$ e $S$ foram particionados, o algoritmo executa a junção sobre as partições utilizando técnicas de *plane-sweeping*.

[1] O universo de um atributo de junção para um conjunto particular de entrada é o menor retângulo cuja cobertura envolve todos os atributos de junção do conjunto de entrada.

Após a realização da fase de filtragem inicia-se a fase de refinamento. A fase de refinamento recebe como entrada tuplas do tipo (*r.id*, *s.id*). Devido ao fato de que as partições não são disjuntas há a possibilidade de haver duplicatas no conjunto de entrada. As duplicatas são eliminadas durante um processo de classificação destas tuplas, que utiliza *r.id* como chave de classificação. A seguir, lê-se do disco tantas tuplas de $R$ quanto possam ser armazenadas na memória, juntamente com vetores contendo pares (*r.id*,*s.id*). Faz-se então os elementos *r.id* apontarem para seus registros correspondentes que estão em memória e classificam-se os elementos de *s.id*, tornando o acesso a $S$ seqüencial. As tuplas de $S$ são lidas para a memória e os atributos de junção de $R$ e $S$ são verificados para determinar se realmente satisfazem o predicado ou não.

Os resultados obtidos pelos autores, sobre dados reais, mostram que o desempenho do PBSM é maior quando nenhuma das entradas estão indexadas ou quando existe um índice sobre a relação menor, do que algoritmos baseados em R-trees. Os algoritmos baseados em R-tree têm melhor desempenho quando há índices disponíveis na relação maior ou quando as duas relações estão indexadas.

Outro algoritmo proposto sobre dois conjuntos não indexados é o apresentado em [APR+98] por Arge *et alii*, denominado SSSJ (*Scalable Sweeping-Based Spatial Join*). Este algoritmo pode ser definido como segue:

```
SSSJ(P, Q: lista de (id,MBR))

   Classifique P e Q em relação ao eixo vertical e atribua a L;
   Execute plane-sweeping;
   Se(não há memória suficente para executar plane-sweeping)
      Junção_Retângulo(L);
      SSSJ(P,Q);
   FimSe
Fim
```

O algoritmo Junção_Retângulo possui os seguintes passos:

1. particiona o espaço em $k$ fatias verticais de forma que no máximo $2N/k$ retângulos iniciem ou terminem em cada fatia;

2. verifica se há retângulos que interceptam mais de uma fatia. Os retângulos que não cabem em apenas uma fatia são particionados em três partes (duas extremidades e uma porção intermediária entre as duas extremidades). A seguir, executa os seguintes passos:

   (a) calcula-se todas as interseções envolvendo porções centrais do conjuntos $P$ e $Q$;

(b) em cada faixa, computa recursivamente todas as interseções que envolvem extremidades dos conjuntos $P$ e $Q$;

3. percorre a lista $L$ em ordem crescente em relação a $y_{min}$. Para cada elemento $r$, os seguintes passos são executados:

   (a) se $r \in P$ e $r$ está contido em uma única faixa $i$, insira $r$ na lista $L_P^{i,j}$. Percorra todas as listas $L_Q^{h,j}$, com $h < i < j$, computando as interseções e removendo da lista todos os elementos que não inteceptam $r$. Escreva $r$ no disco para utilização na recursão referente à faixa $i$;

   (b) se $r \in P$ e $r$ não está contido inteiramente em uma única faixa e sua porção intermediária consiste das faixas $i, i+1, \cdots, j$, então insira $r$ na lista $L_P^{i-1,j+1}$. Percorra todas as listas $L_P^{i',j'}$, com $i < j'$ e $j > i'$, computando as interseções e removendo da lista todos os elementos que não inteceptam $r$. Escreva as extremidades de $r$ no disco para utilização nas recursões apropriadas;

   (c) se $r \in Q$ e $r$ está contido em uma única faixa, repita (3a) com os papéis de $P$

   (d) se $r \in Q$ e $r$ está contido em uma única faixa, repita (3b) com os papéis de $P$.

Os autores investigaram vários algoritmos e variações de *plane-sweeping* aplicados ao SSSJ, além de incluir melhoramentos no PBSM de [PD96]. Os resultados obtidos, utilizando dados reais, foram até 25% melhores que os obtidos com o PBSM original. Entretanto, utilizando métodos de particionamento diferentes do originalmente proposto em [PD96], o PBSM obteve desempenho 10% superior ao SSSJ.

Em [LR96] Lo e Ravishankar propõem a utilização de um método de *hash-join* na computação de junções espaciais, também sobre dados não indexados, denominado *Spatial Hash-Join* (os autores também afirmam que este método pode ser utilizado quando há índices pré-existentes). Este algoritmo, como seu correspondente relacional, é executado em duas fases: uma fase de partição, que separa os objetos em *buckets* utilizando *funções de partição*; e uma fase de junção dos *buckets*.

Os algoritmos de *hash-join* relacional possuem duas restrições, cada uma aplicada a uma fase específica. A primeira, aplicada à fase de partição, é a atribuição de determinado elemento a apenas um *bucket*. A segunda, aplicada à fase de junção, é que cada *bucket* do conjunto interno será comparado com apenas um *bucket* do conjunto externo.

Lo e Ravishankar observam, entretanto, que devido ao fato de que em junções espaciais os predicados são mais complexos do que o de igualdade, pelo menos uma destas restrições deve ser relaxada. Assim, pode-se mapear determinado elemento a vários *buckets* ou comparar cada *bucket* do conjunto interno com vários *buckets* do conjunto externo. Lo e Ravishankar optaram pela segunda alternativa.

O algoritmo inicia calculando o número de partições nas quais o espaço será dividido. Ao contrário do PBSM, estas partições não são regulares. Após definido o número de partições, tomam-se amostras dos dados e os centros dos objetos espaciais contidos nas amostras são utilizados para inicializar as partições. Feito isso, percorre-se o primeiro conjunto de dados, atribuindo os objetos às partições, utilizando uma heurística que procura atribuir os objetos à partição cuja distância do centro do MBR do objeto ao centro da partição for a menor. Ao inserir um objeto em uma partição, o MBR da partição é ajustado para conter o MBR do objeto, o que implica que o centro da partição também pode mudar. Ao fim deste processo, as partições estão formadas e não há replicação nas partições, ou seja, um objeto está contido em apenas uma partição.

O algoritmo continua, agora lendo o segundo conjunto de dados e particionando-o utilizando as mesmas partições utilizadas para ajustar os dados do primeiro conjunto. Nesta parte é permitida a replicação, ou seja, se determinado objeto se sobrepõe a várias partições, o mesmo é armazenado em todas as partições que intercepta. Os objetos que não interceptam nenhuma partição não são considerados.

Após terminar de processar o segundo conjunto de dados, o algoritmo inicia o procedimento de junção. Ele lê uma partição para a memória, constrói uma R-tree sobre os dados e os objetos da segunda são utilizados como janela de consulta sobre o índice.

Este método obteve resultados melhores do que seus concorrentes nos experimentos feitos pelos autores. Os outros métodos investigados foram: construção de uma R-tree com subseqüente construção de uma Seeded Tree; construção de R-trees sobre os dois conjuntos e junção sobre dois índices de R-tree pré-existentes.

Outro trabalho de *hash-join* é apresentado em [dS99] por Silva. O autor propõe alterações nos algoritmos propostos por Lo e Ravishankar [LR96] de forma a minimizar alguns problemas decorrentes desta abordagem. Entretanto, ao contrário de Lo e Ravishankar, da Silva optou por fazer atribuições de objetos a múltiplos *buckets*.

Neste trabalho, da Silva propõe a utilização da *2DHM* (*Two Dimensional Hash Matrix*). A idéia deste método é dividir o espaço em células de tamanho fixo, utilizando uma grade regular. As células são unicamente identificada pelas coordenadas do canto inferior esquerdo. A função de *hash* é aplicada na chave de cada célula que determinado polígono intercepta, produzindo um conjunto de *buckets* onde esse polígono estará armazenado.

Tradicionalmente, o *hashing* espacial de objetos espaciais é feito através da utilização de uma *space-filling curve*, transformando as coordenadas de um espaço $k$-dimensional para um espaço unidimensional e classificando os objetos em relação a esta nova coordenada. Estas "chaves" espaciais são então armazenadas em uma tabela de *hash*. Entretanto, esta abordagem, como visto na seção 2.4.1, não conserva a proximidade espacial nem a extensão do objeto.

Basicamente, a proposta de da Silva é, ao invés de se utilizar uma tabela de *hash*

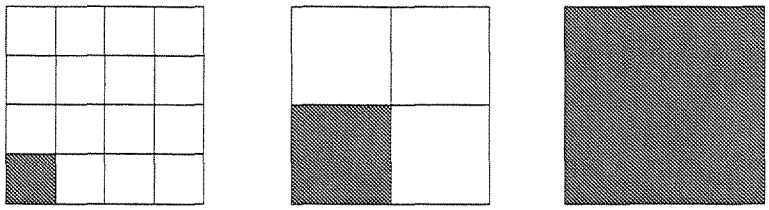



Figura 3.3: Os três tipos de células utilizados na divisão do espaço.

unidimensional, utilizar uma matriz $k$-dimensional de *hash*. Desta forma, procura-se manter a proximidade espacial e extensão original dos objetos na tabela *hash*. Esta matriz não necessariamente teria de ser quadrada, mas poderia obedecer à distribuição dos dados no espaço.

Entretanto, se tomada diretamente, esta proposta possui alguns inconvenientes. O maior deles é o *inchaço* dos dados, ou seja, o número médio de *buckets* em que cada polígono é inserido. O inchaço implica em maior redundância de informação e maior espaço de armazenamento necessário, que pode resultar em maior número de operações de entrada e saída.

A solução adotada pelo autor foi a utilização de células de três tamanhos fixos pre-determinados para o conjunto de dados a partir do tamanho médio dos polígonos. Os tamanhos das células são múltiplos entre si, sendo que as arestas possuem comprimento $2^n$, $n > 0$, e coordenadas iguais a $(x.2^n, y.2^n)$. As células são divididas em três tipos, como mostra a Figura 3.3.

Com esta divisão, apenas células do tipo 2 serão mapeadas para a matriz de *hash*. As células dos tipos 0 e 1 utilizarão a mesma coordenada base da célula de tipo 2 que as contém, sendo mapeadas para a mesma partição e mesmo *bucket*. Isto faz com que um polígono que intercepte uma célula do tipo 2 e alguma células dos tipos 0 e 1 não seja replicado no *bucket* correspondente.

Dentro de cada *bucket*, se houver mais de uma célula do tipo 2, elas estão ordenadas em relação à coordenada base e organizadas hierarquicamente. Isto faz com que, durante a comparação entre *buckets*, apenas os MBRs contidos nas células correspondentes sejam testados. A organização dos *buckets* pode ser melhor visualizada na Figura 3.4.

O algoritmo escolhe para comparação uma combinação de células cuja quantidade de comparações é sempre a mínima possível. A comparação entre células de tipos diferentes também é permitida. A escolha das células que serão comparadas é feita baseada na distribuição dos polígonos através das células tipo 0, 1 e 2. É a escolha de células de tamanho diferentes, valorizando aquelas que possuem menos MBRs a serem comparados,

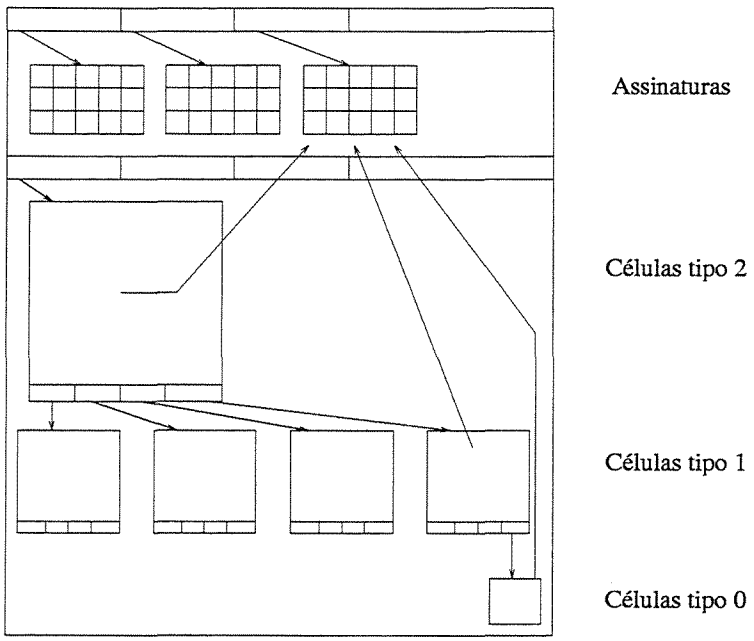



Figura 3.4: Organização de um *bucket.*

que faz com que o número de comparações seja minimizado.

Entretanto, a grande diferenciação deste método está no fato de utilizar uma aproximação *raster* denominada *4CRS* (*four-colour raster signature*), apresentada em [ZdS98] por Zimbrão e de Souza . Segundo os autores, esta aproximação possui as seguintes vantagens: é compacta, possui ótimo desempenho como filtro, além de poder ser utilizada para verificar quais células de um determinado polígono sem a recuperação da geometria completa do polígono. Sua utilização se daria após a fase de junção com MBRs, substituindo as abstrações propostas em [BK94] (veja seção 2.2).

A *4CRS* é uma figura *raster* composta de $m \times n$ células que combina aproximações progressivas e conservativas. Cada célula possui dois bits de informação, que indicam uma das possibilidades apresentadas na tabela 3.1. A Figura 3.5 apresenta um polígono sobreposto por uma grade regular e sua representação *4CRS.*

Para comparar as duas aproximações, basta sobrepô-las e comparar cada célula da primeira aproximação com sua correspondente da segunda aproximação. A decisão de interseção ou não é tomada segundo apresentado na Tabela 3.2. É fácil de se observar que se pelo menos um par de células se intercepta, os polígonos que representam também se interceptam. Apenas em poucos casos as formas dos objetos teriam de ser recuperadas e avaliadas. Entretanto, antes de se comparar duas aproximações deve-se verificar se as células possuem o mesmo tamanho e se suas coordenadas coincidem.

| Tipo | Descrição |
|---|---|
| 00 | A célula não intecepta o polígono |
| 01 | A célula intercepta no máximo 50% do polígono |
| 10 | A célula intercepta mais de 50% do polígono |
| 11 | A célula intercepta o polígono totalmente |

Tabela 3.1: Possíveis valores que uma célula pode assumir em uma assinatura *4CRS*.

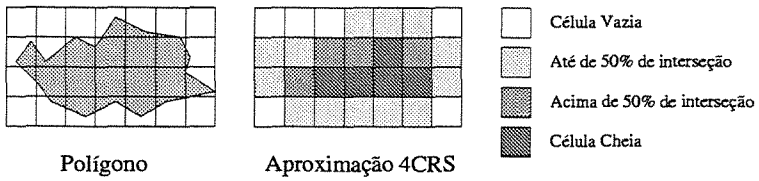



Figura 3.5: Um polígono e sua aproximação *4CRS*.

A proposta de da Silva obteve bom desempenho, principalmente em relação ao número de comparações efetuadas para verificação de interseção. O trabalho ainda inclui boas propostas para a eliminação de duplicatas e para o tratamento de *overflow*.

Koudas e Sevcik, em [KS97b], apresentam um algoritmo para execução de junções espacial derivado do algoritmo de junção da Filter Trees (veja seção 2.4.5), denominado $S^3J$ (*Size Separation Spatial Join*).

Dados dois conjuntos de dados $R$ e $S$, o $S^3J$ constrói partições de Filter Tree sem construir uma Filter Tree inteira. O nível $j$ é composto de $2^j - 1$ linhas igualmente espaçadas em cada dimensão. O nível ao qual um objeto pertence é o maior (ou menor $j$) ao qual o MBR do objeto é interceptado por qualquer linha da grade. Assim, objetos

| Tipo | 00 | 01 | 10 | 11 |
|---|---|---|---|---|
| 00 | Descartada | Descartada | Descartada | Descartada |
| 01 | Descartada | Candidata | Candidata | Aceita |
| 10 | Descartada | Candidata | Aceita | Aceita |
| 11 | Descartada | Aceita | Aceita | Aceita |

Tabela 3.2: Possíveis combinações de valores entre os tipos de células de uma assinatura *4CRS* e sua decisão quanto à interseção.

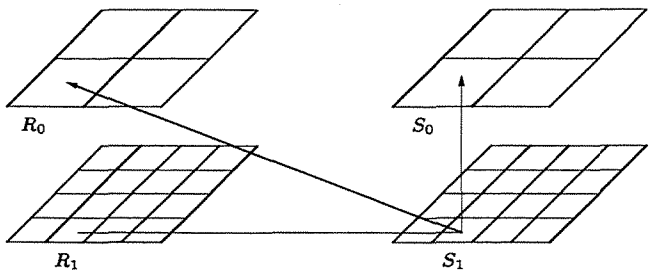



Figura 3.6: Particionamento do espaço e execução de junção segundo o S$^3$J.

com MBRs grandes estarão armazenados em níveis mais altos, enquanto que a maioria dos objetos pequenos estarão armazenados nos níveis mais baixos.

Cada conjunto é lido e particionado em *arquivos de nível*. Para cada objeto, seu nível é identificado, e uma entrada composta por seu MBR, o valor de Hilbert (veja seção 2.4.1) de seu centro e um ponteiro para o dado associado ao objeto é gravada no arquivo de nível correspondente. A seguir, cada arquivo de nível é classificado em ordem crescente dos valores de Hilbert dos objetos e a junção propriamente dita pode iniciar. Seja $R^l(H_s, H_e)$ uma página do $l$-ésimo arquivo de nível $R$ que possui valores de Hilbert variando de $H_s$ a $H_e$ e $L$ o número de arquivos de nível. Então, para todos os níveis $l = 0, \cdots, L$ o algoritmo processa as entradas de $R^l(H_s, H_e)$ com aquelas contidas em $S^{l-i}(H_s, H_e)$ para $i = 0, \cdots, l$ e as entradas em $S^l(H_s, H_e)$ com aquelas em $R^{l-i}(H_s, H_e)$ para $i = 1, \cdots, l$. Isto pode ser melhor visualizado na Figura 3.6: os elementos contidos na partição $R_1$ serão processados apenas em relação aos elementos das partições $S_1$ e $S_0$. Por sua vez, os elementos de $S_1$ serão processados apenas em relação aos elementos da partição $R_0$.

O S$^3$J foi comparado em relação ao PBMS [PD96] e ao *Spatial Hash Join* [LR96], mostrando-se melhor tanto em resultados experimentais quando em resultados de modelos analíticos apresentados pelos autores.

### 3.3.2 Modelos analíticos de desempenho para junções espaciais

Os primeiros modelos analíticos de desempenho de métodos de acesso espaciais surgiram bem depois das primeiras propostas. O mesmo não aconteceu em relação às propostas de junção espacial. Pode-se atribuir este fato à necessidade de uma melhor compreensão desta operação, bem como à necessidade de se integrar métodos eficientes de otimização de consultas, que fazem uso de modelos analíticos para escolher a melhor forma de se executar determinadas consultas. Os principais modelos analíticos de junção espacial concentram-se sobre as R-trees e são apresentados em [Gün93, TSS97, TSS98].

Em [Gün93], Günther propõe o primeiro modelo analítico para junções espaciais, além

| Símbolos | Definição |
|---|---|
| $n$ | altura média da árvore |
| $k$ | quantidade média de filhos de um nó da árvore |
| $s$ | tamanho da página, em bytes |
| $M$ | tamanho da memória principal, em páginas |
| $\pi_{i,j}$ | probabilidade de que $o_1 \Theta o_2$ seja verdadeiro, dado $o_1$ no nível $i$ e $o_2$ no nível $j$ |
| $C_\Theta$ | tempo gasto para se verificar se $o_1 \Theta o_2$ |
| $C_{I/O}$ | tempo gasto para executar uma operação de I/O |
| $m$ | número médio de tuplas armazenadas em uma página de disco |

Tabela 3.3: Parâmetros e variáveis utilizados no modelo proposto por Günther.

de propor modelos analíticos para inserção (atualização) e consultas de seleção (*range queries*). Os modelos desenvolvidos foram sobre as três técnicas estudadas em seu trabalho, a saber: *nested loop*, junção utilizando *Generalization Trees* e *join indices*.

Günther considera o custo da junção espacial em *Generalization Trees* como uma série de operações de *range queries* do primeiro conjunto sobre o segundo e vice-versa. A função, com pequenas diferenças entre os casos de dados "clusterizados" ou não, divide-se em duas partes: o custo da computação da comparação e o custo das operações de entrada/saída (I/O). Algumas das variáveis consideradas no modelo analítico de Günther são mostradas na Tabela 3.3.

A computação dos custos de comparação leva em consideração a distribuição dos objetos no espaço, o que faz com que seja necessário uma estimativa da probabilidade de que dois elementos $a$ e $b$ se combinem. Esta probabilidade, que varia de acordo com o tipo de distribuição dos dados e ao nível onde se encontram os elementos, é dada por $\pi_{i,j}$, onde $i$ e $j$ são os níveis onde se encontram os elementos. Assim, o custo de computação do predicado é dado por:

$$D_\Theta = C_\Theta . \sum_{i=0}^{n} (\pi_{i,i-1} k^{2i})(1 + \sum_{j=i}^{n-1} (\pi_{i,j} + \pi_{j,i}) k^{j-i+1})$$

Com relação aos custos de I/O, sabe-se que apenas aqueles nós que satisfazem o predicado deverão ser acessados. Assim, uma estimativa da quantidade de nós participantes, incluindo a raiz, da junção é dada por $1 + \sum_{i=0}^{n-1} (\pi_{i,0} k^{i+1})$ e $1 + \sum_{i=0}^{n-1} (\pi_{0,i} k^{i+1})$, para as árvores A e B, respectivamente.

Sendo que a memória principal possui $M$ páginas, pode-se preencher $M - x$ com os nós superiores de uma das árvores e efetuar a busca na outra árvore. Após esta

primeira passagem, pode-se carregar os nós restantes e continuar a busca. No total haverá $\lceil \frac{1+\sum_{i=0}^{n-1} \pi_{i,0} k^{i+1}}{m.(M-x)} \rceil$ passagens.

Seja $Y(x, y, z)$ o número de operações de I/O necessárias para acessar $x$ registros aleatoriamente em um arquivo de $z$ registros armazenados em $y$ páginas de disco. Então $Y$ pode ser definido como:

$$Y(x, y, z) = y. \left[ 1 - \prod_{i=1}^{x} \frac{z - z/y - i + 1}{z - i + 1} \right]$$

Então, em cada passagem são executadas $\sum_{i=0}^{n-1} Y(\lceil \pi_{0,i} k^{i+1} \rceil, \lceil \frac{N}{m} \rceil, N)$ acessos a disco quando os dados não estão clusterizados e $\sum_{i=0}^{n-1} Y(\lceil \pi_{0,i} k^{i} \rceil, \lceil \frac{k^{i+1}}{m} \rceil, k^i)$ quando os dados estão clusterizados.

Combinando estas funções, temos que o custo total de I/O para dados clusterizados é aproximadamente

$$D_{I/O} = C_{I/O}. \left[ \lceil \frac{1 + \sum_{i=0}^{n-1} \pi_{i,0} k^{i+1}}{m.(M-x)} \rceil . \sum_{i=0}^{n-1} Y(\lceil \pi_{0,i} k^{i+1} \rceil, \lceil \frac{N}{m} \rceil, N) + \sum_{i=0}^{n-1} Y(\lceil \pi_{0,i} k^{i+1} \rceil, \lceil \frac{N}{m} \rceil, N) \right]$$

enquanto que, para dados não clusterizados, é aproximadamente.

$$D_{I/O} = C_{I/O}. \left[ \lceil \frac{1 + \sum_{i=0}^{n-1} \pi_{i,0} k^{i+1}}{m.(M-x)} \rceil . \sum_{i=0}^{n-1} Y(\lceil \pi_{0,i} k^{i} \rceil, \lceil \frac{k^{i+1}}{m} \rceil, k^i) + \sum_{i=0}^{n-1} Y(\lceil \pi_{0,i} k^{i} \rceil, \lceil \frac{k^{i+1}}{m} \rceil, k^i) \right]$$

Dadas estas funções, temos que o custo de uma junção espacial, em uma estrutura do tipo da *generalization tree* é:

$$D = D_{\Theta} + D_{I/O}$$

Outro modelo analítico de desempenho é apresentado por Theodoridis *et alii* em [TSS97, TSS98]. O objetivo é obter uma fórmula que estime a quantidade de acessos a disco ($DA$) necessários para realizar a junção espacial entre dois conjuntos indexados em R-trees, baseando-se no *número de objetos* e na *densidade do espaço de trabalho*. Novamente aqui, a junção espacial é considerada como uma seqüência de operações de *range queries*. Os algoritmos básicos utilizados são os propostos por Brinkhoff *et alii* em [BKS93]. A Tabela 3.4 apresenta os elementos considerados no modelo.

De acordo com o trabalho apresentado em [TS95], a fórmula que estima o custo de uma *range query* $q = (q_1, \cdots, q_n)$ sobre um conjunto de dados contendo $N$ objetos com densidade $D$, é dado por

$$DA(q) = \sum_{j=1}^{h-1} \left\{ N_j . \prod_{i=1}^{n} (s_{j,i} + q_i) \right\} \tag{3.1}$$

| Símbolos | Definições |
|---|---|
| $n$ | número de dimensões |
| $m$ | número mínimo de elementos que um nó pode conter |
| $m$ | número máximo de elementos que um nó pode conter |
| $c$ | ocupação média dos nós da estrutura, em % |
| $s_{j,i}$ | extensão média de um nó do nível $j$ na dimensão $i$ |
| $h_R$ | altura da árvore $R$ |
| $N_R$ | quantidade de objetos de dados indexados em $R$ |
| $D_R$ | densidade dos dados indexados por $R$ |
| $N_{R,j}$ | número de nós da árvore $R$ no nível $i$ |
| $D_{R,i}$ | densidade dos retângulos dos nós de $R$ no nível $i$ |
| $S_{R,j,k}$ | tamanho médio dos retângulos dos nós da árvore $R$ no nível $j$ na dimensão $k$ $(k = 1, \cdots, n)$ |
| $DA(R,S)$ | número de acessos a disco para uma junção espacial entre as árvores $R$ e $S$ |

Tabela 3.4: Parâmetros e variáveis utilizados no modelo proposto por Theodoridis *et alii.*

ou seja, o número de acessos a disco $DA$ é igual à soma da cobertura todal, em cada nível $j$, dos nós $s$ da R-tree, assumindo que seu tamanho foi extendido pelo tamanho da janela de consulta $q$ em cada dimensão $i$. O parâmetro $s_{i,j}$ é dado em função da densidade[2] dos dados no nível $j$ e é definido por

$$s_{j,i} = \left(\frac{D_j}{N_j}\right)^{\frac{1}{n}}$$

Considerando que os dois índices têm a mesma altura $h$ e que as raizes estão armazenadas na memória, no nível $j = h - 1$, a árvore $R$ $(S)$ contém $N_{R,j}$ nós $E_{R,j,k}$, $1 \leq k \leq N_{R,j}$ $(E_{S,j,k},\ 1 \leq k \leq N_{S,j})$ de tamanho médio igual à $s_{R,j}$ $(s_{s,j})$. Para determinar quais elementos se combinam, comparam-se as entradas de $E_{R,j,k}$ com as de $E_{S,j,k}$.

O custo das comparações citadas é denotado por $DA(R,S,j)$, que corresponde à soma de dois fatores que expressam o custo para as duas R-trees:

$$DA(R,S,j) = (custo_{R,j}) + (custo_{S,j}) \tag{3.2}$$

onde

[2]A *densidade* de um conjunto de $N$ objetos é dada pela soma das áreas dos objetos dividida pela área do espaço de dados.

$$custo_{R,j} = \sum_{N_{S,j}} intsect(N_{R,j}, s_{R,j}, s_{S,j}) \tag{3.3}$$

e

$$custo_{S,j} = N_{S,j} + \sum_{N_{S,j}} intsect(N_{R,j+1}, s_{R,j+1}, s_{S,j}).intsect(1, s_{R,j+1}, s_{S,j}) \tag{3.4}$$

onde $intsect(N_j, s_j, q)$ é uma função que retorna o número de nós do nível $j$ interceptados pela janela de consulta $q$, sendo definida por:

$$intsect(N, s, q) = N. \prod_{k=1}^{n} (s_k + q_k) \tag{3.5}$$

Estas fórmulas têm o seguinte significado: assumindo que as entradas de $R$ ($S$) no nível $j$ funcionam como um conjunto de dados (um conjunto de janelas de consulta), aplica-se a função *intsect* (derivada da função 3.1) para estimar o custo de acesso da árvore $R$, enquanto que, para estimar o custo de acesso à árvore $S$, adicionamos ao custo de leitura dos nós de $S$ no nível $j$ um segundo fator, que calcula o número estimado de nós de $R$ no nível $j+1$ que interceptam nós de $S$ no nível $j$, multiplicado pela probabilidade da existência da interseção destes nós.

Combinando as fórmulas de 3.2 a 3.5, $DA(R, S, j)$ é dado por:

$$DA(R,S,j) = N_{S,j} \left\{ N_{R,j} \prod_{k=1}^{n} (s_{R,j,k} + s_{S,j,k}) + 1 + N_{R,j+1} \left\{ \prod_{k=1}^{n} (s_{R,j+1,k} + s_{S,j,k}) \right\}^2 \right\} \tag{3.6}$$

Processando a junção em cada nível das duas árvores até as folhas, o custo total da junção $DA(R, S)$ para os $j$ níveis é igual a

$$DA(R,S) = \sum_{j} DA(R,S,j) \tag{3.7}$$

A equação 3.7 é sensível à ordem dos dois índices: uma vez que a junção espacial é considerada como uma seqüência de operações de *range queries* de um conjunto de dados sobre o outro, o desempenho é melhor se as janelas de consultas forem tomadas do conjunto menor sobre o conjunto maior.

Sejam $h_R$ e $h_S$, $h_R > h_S$ as alturas das árvores $R$ e $S$, respectivamente. Para o caso de quando as árvores têm alturas diferentes, ainda é possível utilizar a fórmula dada em 3.6, com pequenas modificações, por que o desempenho é o mesmo nos $h_R - h_S$ primeiros níveis. Se $j'$ é a variável que indica o nível atual relacionado à árvore de menor altura, com

o nível das folhas correspondendo ao nivel 1, a quantidade de acessos a disco $DA'(R,S,j)$ é dada por

$$DA'(R,S,j) = N_{S,j'}\left\{N_{R,j}\prod_{k=1}^{n}(s_{R,j,k}+s_{S,j',k})+1+N_{R,j+1}\left\{\prod_{k=1}^{n}(s_{R,j+1,k}+s_{S,j',k})\right\}^2\right\}$$

onde

$$j' = \begin{cases} j-(h_R-h_S), & h_R-h_S+1 \le j \le h_R-1 \\ 1, & 1 \le j \le h_R-h_S \end{cases}$$

Theodoridis *et alii* apresentam resultados comparativos entre os modelos analíticos e execuções reais dos algoritmos, baseando-se em dados reais e dados sintéticos. Os resultados obtidos mostram que o modelo analítico obteve respostas muito aproximadas em relação às obtidas em execuções reais, com diferença em torno de 15% em relação ao obtido experimentalmente e o obtido analiticamente.

### 3.3.3 Outras pesquisas sobre junções espaciais

A área de junções espaciais ainda é um campo muito fértil para pesquisas. Dentre estas pesquisas destacamos o *benchmark* de junções espaciais, algoritmos especiais para o tratamento de múltiplas entradas na operação de junção. Nesta seção veremos alguns destes trabalhos, para se ter uma idéia das novas linhas de pesquisa nesta área.

*Benchmarks*

Trabalhos de *benchmarking* de junções espaciais são encontrados em [GOP+98], de Günther *et alii*, e em [HS95], de Hoel e Samet.

Hoel e Samet [HS95] fazem a comparação entre métodos da família das R-trees (R-tree linear, R-tree quadrática, R+-tree, R*-tree) e de uma estrutura denominada *PMR quadtree*. Os métodos são avaliados em relação ao algoritmo proposto em [BKS93]. Os testes foram efetuados sobre dados reais, colhendo resultados de desempenho em relação a operações de I/O e tempo de CPU. Os autores também investigam o desempenho quando a junção resulta na geração de um mapa de saída, contendo todos os pontos de interseção, e não apenas quando o resultado da junção é uma lista de tuplas contendo identificadores dos objetos que se interceptam, o caso mais comum.

Günther *et alii* [GOP+98] apresentam uma ferramenta *WWW* para produção de MBRs de acordo com parâmetros especificados pelo usuário, incluindo várias distribuições de dados (uniforme, normal e exponencial). A Tabela 3.5 apresenta os parâmetros que podem ser controlados para a geração destes dados. A página *WWW* do gerador é `http://www.enst.fr/~bdtest/sigbench/menu.html`.

| Parâmetro | Significado |
|---|---|
| $C$ | cobertura; a razão entre a soma das áreas dos retângulos e a área do conjunto universo |
| $x$ | coordenada $x$ do canto inferior esquerdo do retângulo |
| $y$ | coordenada $y$ do canto inferior esquerdo do retângulo |
| $t$ | inclinação da diagonal principal do retângulo |
| $a$ | área do retângulo |
| $N$ | número de retângulos |
| $x_{min}$ | a menor coordenada $x$ possível |
| $y_{min}$ | a menor coordenada $y$ possível |
| $x_{max}$ | a maior coordenada $x$ possível |
| $y_{max}$ | a maior coordenada $y$ possível |
| $U$ | área do universo |

Tabela 3.5: Parâmetros do gerador de retângulos.

Nesse trabalho, foram implementados os métodos de junção *Nested loop*, *Scan-and-Index* e *Synchronized Tree Traversal*, já discutidos aqui, neste capítulo. O ambiente de testes utilizado foi o Sistema Gerenciador de Banco de Dados Orientado a Objetos $O_2$.

Papadopoulos *et alii* [PRS99]também propõem um *benchmark* para a análise de desempenho de vários métodos de junção espacial. Sua intenção é desenvolver uma arquitetura única para a avaliação de vários métodos de acesso espaciais com relação à consultas e junções espaciais. Esta arquitetura possui módulos comuns a todos os métodos, como exemplo o módulo responsável pelas operações de I/O e o gerenciador de *buffer*. Além disso, os autores utilizam um processador de consultas, que se utilizam de modelos de custo para a otimização dessas consultas.

### Processamento de junção com múltiplas entradas

Em sistemas gerenciadores de bancos de dados, tanto espaciais quanto convencionais, é comum a utilização de consultas que envolvem o processamento de vários conjuntos de dados. Consultas do tipo "Quais prédios públicos estão próximos a praças vizinhas de supermercados?" envolvem a determinação de um bom plano de execução, que minimize o tempo de avaliação e a utilização de recursos e um mecanismo de execução que aplique este plano, gerenciando o sincronismo entre os operadores.

Junções que consideram múltiplas entradas podem ser expressas como um grafo $Q = (V, E)$, onde cada vértice $V$ corresponde a uma relação espacial e cada aresta $E$ corresponde a um predicado de junção. A consulta pode ser executada utilizando várias

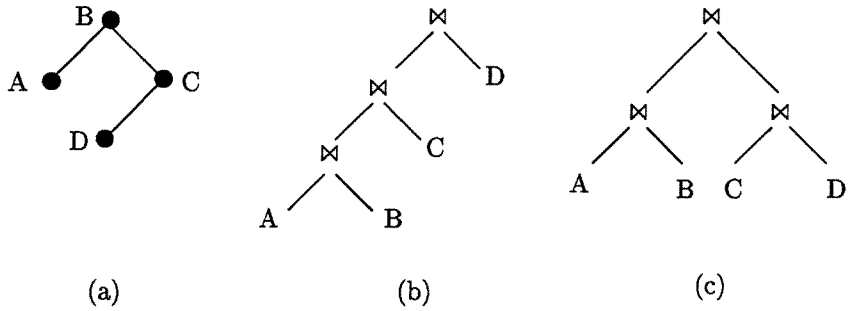



Figura 3.7: Possíveis modos de processamento de uma consulta representada por um grafo (a): *left-deep plan* (b) e *bushy plan* (c).

combinações diferentes de algoritmos de junção sobre a as relações, tomadas duas a duas. Podemos visualizar estas diferentes combinações na Figura 3.7.

Muitas técnicas foram desenvolvidas para a otimização e execução de consultas relacionais. Técnicas utilizadas para consultas relacionais, entretanto, não são aplicáveis diretamente a operações sobre dados espaciais. Isto se deve à não transitividade do principal predicado utilizado em consultas espaciais: a *interseção*.

Entretanto, a grande maioria dos trabalhos relacionados com junções espacias está centralizada no tratamento de junções para duas entradas apenas. Exceções são os trabalhos apresentados em [PMT99] por Papadias, Mamoulis e Theodoridis, e em [MP99] por Mamoulis e Padapadias.

Em [MP99], Mamoulis e Papadias adaptam um conjunto de operadores utilizando métodos baseados em métodos STT (*synchronized tree traversal*), em métodos de *hash-join*, e em um método de sua proposta, o SISJ (veja seção 3.3.1). Além de propor os planos de execução, os autores introduzem análises de custo, um algoritmo que retorna um plano de execução ótimo baseando-se nas fórmulas de custo e no grafo da consulta, e avaliações de desempenho utilizando dados reais e sintéticos.

Em [PMT99], Papadias, Mamoulis e Theodoridis propõem a utilização de um tipo de grafo denominado CSP (*constraint satisfaction problem*) binário. Este tipo de grafo é definido por:

- um conjunto de $n$ variáveis $v_1, v_2, \cdots, v_n$;
- um domínio finito $D_i = (u_{i,1}, u_{i,2}, \cdots, u_{i,N_i})$, para cada variável $v_i$, com $\mid D_i \mid = N_i$;
- uma restrição binária $C_{ij} \in D_i \times D_j$ para cada par de variáveis $v_i, v_j$;

Os autores propõem um algoritmo denominado *Hybrid*, uma combinação de grafos CSP com duas metodologias de busca em R-trees, a *window reduction* (WR), que nada

mais é do que uma seqüência de janelas de consultas tomadas de um conjunto sobre outro, e a *synchronous traversal*, que é uma generalização de métodos STT para várias entradas.

A idéia é utilizar os recursos dos métodos STT para definir uma área de interseção entre as várias entradas, para então instanciar as variáveis que serão utilizadas pelo método WR. A utilização dos grafos seria nas operações de *forward* e de *backtracking* na árvore de planos de execução.

Além de definir este algoritmo, os autores apresentam modelos analíticos de custo e um algoritmo para a geração de um plano de execução ótimo, baseados nestes modelos de custo.

### Trabalhos sobre a fase de refinamento

Os passos de filtragem e refinamento são uma estrutura já bem estabelecida para o processamento de consultas espaciais. Mas, ao contrário da fase de filtragem, a fase de refinamento tem recebido pouca atenção dos esforços de pesquisa, embora também seja uma fase crítica em relação ao desempenho. Exceções à regra são os trabalhos de Patel e DeWitt [PD96] (veja seção 3.3.1) e mais recentemente de Abel *et alii* [AGPZ99].

O trabalho de Abel *et alii* concentra-se exclusivamente na fase de refinamento. Nesta proposta, além do *buffer* do sistema, responsável pelas trocas de páginas em operações de I/O, há um outro *buffer* denominado *aplication buffer*, onde são mantidos os objetos recuperados através do identificador obtido na fase de filtragem.

Sua proposta também investiga a ordenação na qual os objetos serão comparados, ou seja, dada uma lista $F$ de pares de elementos $(r_i, s_j)$, obtida na fase de filtragem, qual será a ordem de leitura destes elementos na fase de refinamento. Quatro opções de sequenciamento foram avaliadas:

- *naive*: na qual os objetos são lidos de acordo com a seqüência obtida da fase de filtragem;
- simples: os elementos são ordenados de acordo com o identificador dos objetos. Duplicatas são eliminadas durante a ordenação;
- segmentada: os elemento de $F$ são ordenados em relação a $r_i$. Então um subconjunto $R_s$ da relação $R$ é carregado no *aplication buffer*, seguida da classificação de um subconjunto de $F$, que contém elementos da relação $S$ que combinam com $R_s$. Os pares são então buscados no disco e suas descrições são passadas para o refinamento. Este método privilegia um conjunto em detrimento de outro.
- *zig-zag*: proposta pelos autores, os elementos são classificados, e lidos para o *aplication buffer*. Então o algoritmo alterna o foco em relação a um ou a outro conjunto de dados.

A estratégia *zig-zag* foi validada através de experimentos utilizando dados reais sob diversas configurações, utilizando métodos da família das R-tree como índices, e provou-se ter bom desempenho médio no seqüenciamento de pares de elementos resultantes da fase de filtragem.

# Capítulo 4

# Implementação e Resultados

## 4.1 Introdução

Nos capítulos anteriores, apresentamos vários métodos de acesso espaciais, bem como vários métodos de junção espacial. Através deste estudo prévio, procuramos determinar os métodos de junção mais apropriados à implementação e análise.

O método escolhido para servir de índice foi a R*-tree [BKSS90], reconhecidamente o método mais robusto e eficiente dentre os membros da família das R-trees, talvez perdendo apenas para a Hilbert R-tree [KF94]. Entretanto, uma vez que a R*-tree é um método largamente utilizado nas análises de desempenho da área de métodos de acesso espaciais, decidimos executar os testes de desempenho sobre esta estrutura de indexação.

Os métodos de junção escolhidos para avaliação foram os baseados em busca sincronizada apresentados em [BKS93] e em [HJR97]. Isto se deve à presença de estruturas da família das R-trees presentes em grande parte dos sistemas que lidam com dados espaciais. Além disso, mesmo quando não há índices disponíveis, ainda é possível a construção em tempo de execução. A existência de procedimentos e técnicas que realizam inserções intensivas (seções 2.7.5 e 2.7.6) faz com que seja interessante a investigação da viabilidade da construção de índices sobre dados não indexados e em seguida realizar a junção, ao invés de se utilizar um método de junção que recebe como entrada dados não indexados.

Este capítulo está organizado da seguinte forma: a seção 4.2 descreve os critérios utilizados na análise dos métodos de junção espacial; a seção 4.3 descreve a arquitetura de nossa implementação; a seção 4.4 descreve os dados utilizados na construção dos índices; a seção 4.5 descreve as variações utilizadas nas execuções dos procedimentos de junção; e finalmente, a seção 4.6 mostra os resultados obtidos nos experimentos, bem como uma discussão sobre esses resultados.

## 4.2 Os critérios de análise

A análise analítica ou não analítica de métodos envolve dois tipos de complexidade: a complexidade de tempo e a complexidade de espaço. A primeira está relacionada com o tempo demandado pelo método para a sua execução. A segunda envolve os requisitos de espaço do método, seja em memória principal ou em disco.

A ênfase deste experimento se dará em relação aos requisitos de tempo. Isto se deve ao fato de que, em sistemas interativos como sistemas gerenciadores de bancos de dados (SGBDs) ou sistemas de informações geográficas (SIGs), um dos requisitos satisfação do usuário é o *tempo* que determinada operação leva até sua conclusão. Isto sem considerar que, quanto mais rapidamente uma operação for executada, mais rapidamente o processo que a requisitou liberará os recursos a ele alocados.

Operações típicas de SGBDs e SIGs envolvem grandes quantidades de acessos a disco. Estas operações de I/O são extremamente caras se comparadas às instruções executadas pela CPU. Enquanto estas estão na casa dos milissegundos, aquelas estão na faixa dos microssegundos. Se considerarmos que uma operação de junção ou outra consulta qualquer pode acessar todas as páginas de determinado índice, e que este índice pode compreender milhares de páginas, uma execução ineficiente demandará altos custos em relação ao tempo de execução dessa consulta.

Desta forma, neste trabalho, o tempo será dado em função da quantidade de páginas requisitadas do disco. Embora também analisemos o tempo de CPU gasto pelos métodos implementados, ressaltamos que nos métodos não foram feitas otimizações neste sentido, devendo ser levada em menor consideração. Com relação à quantidade de acessos a disco, esta mereceu mais atenção, de forma que procuramos maximizar o desempenho dos algoritmos neste quesito.

Em especial, estamos interessados em estudar a influência que o tamanho de página, o tamanho do *buffer* e as políticas de trocas no *buffer* têm no desempenho dos algoritmos de junção espacial.

## 4.3 A arquitetura do sistema

Para facilitar a avaliação dos métodos de junção espacial, foi implementado um sistema com uma arquitetura modularizada, mostrado na Figura 4.1. Assim, temos o módulo de abstração do disco, o módulo de junção, que contém os métodos de junção espacial, o método de acesso, módulo responsável pela indexação dos dados, além de módulos com funções específicas. Os módulos foram implementados na linguagem C e executados numa máquina SUN SparcStation 20 com 128 megabytes de memória, rodando o sistema operacional Solaris.

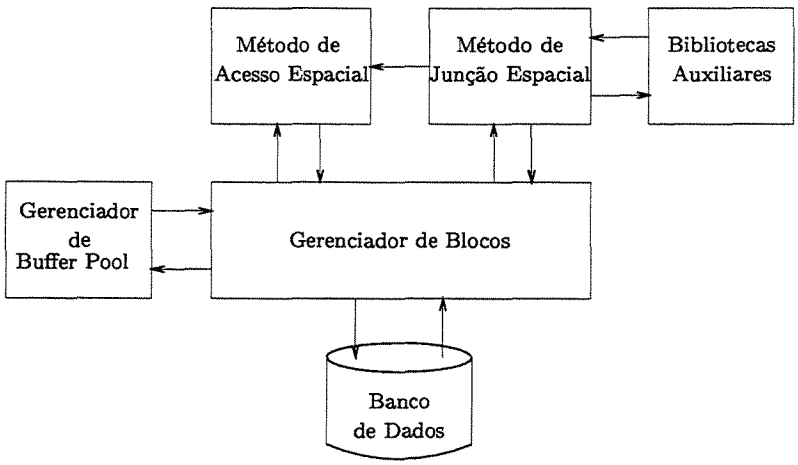



Figura 4.1: A arquitetura do sistema.

Para manter a compatibilidade com os trabalhos de Cox [CM92] e Carneiro [Car98], reutilizamos o módulo de indexação e, com algumas modificações, o módulo de abstração de disco.

### 4.3.1 Módulo Gerenciador de Blocos

O módulo gerenciador de blocos é o responsável pela interface entre o disco e os módulos de junção e de métodos de acesso. Este módulo, inicialmente implementado por Cox, foi criado com o intuito de dar o controle ao pesquisador sobre as operações de entrada e saída. Foram necessárias algumas alterações neste módulo, de forma a adequá-lo ao foco deste trabalho. Entretanto, estas alterações, detalhadas mais adiante, não afetam o comportamento inicialmente proposto por Cox.

Este módulo é o responsável pelas operações de entrada e saída do sistema. Todos os pedidos de leitura e escrita são feitos através dele. Como a intenção é aproximar-se o mais possível das condições existentes num SGBD/SIG, junto a este módulo foi implementado um *bufferpool*. Páginas lidas do disco são armazenadas neste *bufferpool*, da mesma forma que páginas escritas no disco. Este módulo também possui rotinas responsáveis pela contagem de páginas lidas ou escritas, *hits* e *misses* do *bufferpool*, dentre outros fatores.

A quantidade de páginas que serão mantidas no *bufferpool* é um parâmetro de entrada da operação de junção, bem como a política de substituição de páginas. Os métodos de substituição de páginas estão implementados em um módulo em separado, acessado apenas pelo Gerenciador de Blocos, o *Gerenciador de bufferpool*. A implementação inicial suportava apenas o método LRU (*least recently used*). Foram incluídos ainda o método

LFU (*least frequently used*), o método de substituição aleatória (*random*), e método SC (*second chance*), uma versão mais otimizada do FIFO (*first in first out*).

Outro fator configurável a partir de um parâmetro de entrada da operação é a quantidade de páginas a ser utilizada para *buffer pinning*. O *buffer pinning* consiste na fixação de páginas no *bufferpool*, até que todas as referências a ela tenham sido processadas. Isto evita que uma página muito referenciada seja descartada pela política de substituição de páginas. O *buffer pinning* também não constava na implementação original de Cox e Carneiro, sendo incluído neste trabalho.

As páginas que serão fixadas no *bufferpool* são escolhidas através de um conjunto de funções que calcula o grau de interseção existente entre os elementos de uma lista. A lista de entrada para estas funções é a lista de elementos obtidos da junção entre dois ou mais nós. Assim, cada vez que uma página vai ser lida, verifica-se se esta página é uma das que devem ser fixadas. Se for, ela é fixada no *bufferpool* e lá permanece até que seja dada a ordem de liberação desta página. Caso ocorra um *miss* e todas as páginas do *buffer pool* estejam fixadas, escolhe-se uma página para substituição conforme a política de substituição escolhida pelo usuário.

A cada leitura de página, seu grau é diminuído em uma unidade, até chegar a zero. Isto significa que esta página não será mais utilizada, podendo ser eliminada da lista de graus e do *bufferpool*, se estava fixada.

### 4.3.2 Módulo de métodos de acesso espacial

Este módulo é o responsável pela inserção de elementos no índice, remoção, consulta e inicialização. Além disso, o método de acesso espacial mantém os parâmetros como tamanho do bloco utilizado, números máximo e mínimo de entradas permitidos em cada nó, número de dimensões, dentre outros parâmetros.

As raízes das árvores são mantidas na memória. Além das raízes, mantemos também páginas que contêm os nós correntes de cada árvore. Estas páginas são mantidas para melhorar o desempenho e para haver uma maior simplificação dos algoritmos.

A operação de junção utiliza este módulo para a inicialização e manutenção dos parâmetros da estruturas de índices e das variáveis inseridas na chamada da junção. Em nosso trabalho, o método de acesso utilizado é a R*-tree, embora a variações de R-tree original também possam ser utilizadas sem modificação nos parâmetros da operação de junção. Com pequenas modificações é possível incorporar a R+-tree. Entretanto, como a R*-tree é o método mais comum e de melhor desempenho geral, preferimos nos ater apenas à R*-tree.

Maiores detalhes deste módulo são dados nos trabalhos de Cox [CM92] e Carneiro [Car98].

### 4.3.3 Módulo de métodos de junção espacial

É no módulo de métodos de junção espacial que se encontra a maior parte de nosso trabalho. Este módulo constitui-se de três métodos de junção espacial: o *nested loop* (NL) com entradas indexadas, o método *depth-first* (junção em profundidade – DF) com as otimizações propostas por Brinkhoff [BKS93], e o método *breadth-first* (junção em largura – BF) proposto por Huang *et alii* [HJR97].

O algoritmo básico da junção espacial em largura pode ser visto como uma seqüência de operações de *range queries*. E como tal, pode ser feito em profundidade ou em largura. O algoritmo de junção em profundidade, para árvores de mesma altura, pode ser definido como segue:

```
Junção_em_profundidade(R, S: nó)

   Para (∀ r ∈ R) Faça
      Para (∀ s ∈ S | r.MBR ∩ s.MBR ≠ ∅) Faça
         Se (R é página) Então
            Inclua (r, s) no conjunto-resposta;
         Senão
            Leia(r.ponteiro);
            Leia(s.ponteiro);
            Junção_em_profundidade(r.ponteiro, s.ponteiro);
         FimSe
      FimPara
   FimPara
Fim
```

Este algoritmo, denominado *nested-loop* (NL) foi o primeiro a ser implementado neste trabalho e servirá como meio comparador para as medidas de desempenho.

Brinkhoff observa que apenas os elementos cujos MBRs estão na interseção dos MBRs de seus pais podem participar da operação de junção. Sendo assim, o algoritmo sofre uma pequena modificação. Antes de iniciar, calcula-se o MBR correspondente à interseção entre os dois nós, e eliminam-se de ambos os elementos que não interceptam a área de sobreposição entre os nós pais. Isto faz com que o número de comparações seja diminuído, poupando tempo de CPU.

O novo algoritmo então seria como segue:

```
Junção_em_profundidade(R, S: nó; Intr:  MBR)
   // Intr é o MBR correspondente à interseção de R e S
```

```
   Elimine todas as entradas r ∈ R | r.MBR ∩ Intr = ∅;
   Elimine todas as entradas s ∈ S | s.MBR ∩ Intr = ∅;
   Para (∀ r ∈ R) Faça
      Para (∀ s ∈ S | r.MBR ∩ s.MBR ≠ ∅) Faça
         Se (R é página) Então
            Inclua (r, s) no conjunto-resposta;
         Senão
            Leia(r.ponteiro);
            Leia(s.ponteiro);
            Junção_em_profundidade(r.pointeiro, s.ponteiro, r.MBR ∩ s.MBR);
         FimSe
      FimPara
   FimPara
Fim
```

A eliminação de elementos que não fazem parte da interseção, dependendo do conjunto de entradas, diminui consideravelmente o número de comparações. Sejam R e S os nós que estão sendo comparados no momento, tal que $n$ e $m$ sejam o número de entradas de R e S respectivamente. Pela primeira abordagem, teríamos que efetuar $m \times n$ comparações. A eliminação de entradas de R e S exige $n + m$ comparações. O pior caso é se não conseguirmos eliminar nenhuma entrada, pois nesta passagem teríamos que efetuar $n + m + (n \times m)$ comparações. Mas basta eliminar um pequeno número de entradas que o esforço será compensador.

Esta eliminação será útil na preparação para outro melhoramento do algoritmo: a incorporação de *plane-sweeping*. Esta é uma técnica de Geometria Computacional comum de se computar interseções. O algoritmo implementado é o proposto em [BKS93]. Um pré-requisito do algoritmo é que as entradas dos nós estejam classificadas em relação à coordenada $x$ do canto inferior esquerdo.

A idéia básica deste algoritmo é mover uma linha, perpendicular a um dos eixos das dimensões, da esquerda para a direita, verificando interseções em relação à projeção dos MBRs sobre um dos eixos. São tomados elementos dos dois conjuntos alternada e sincronizadamente. Ao escolher determinado elemento de um conjunto, o algoritmo irá computar todas as interseções deste elemento com os elementos do segundo conjunto. Uma vez que os nós estão classificados em relação à coordenada $x$ mínima, o algoritmo toma vantagem desta característica para não ter que percorrer os conjuntos do início até o fim, reduzindo o número de comparações.

O algoritmo pode ser formalizado como segue:

```
TesteInterseção(R, S: nó; lista_de_pares:  lista de ((id, MBR),(id, MBR)))
```

```
// || R || e || S || são as cardinalidades de R e S respectivamente
Var i, j, lista_de_pares;

    i := 1;
    j := 1;
    Enquanto (i ≤ || R ||) ∧ (j ≤ || S ||) Faça
       Se (R[i].x_min < S[j].x_min) Então
          LoopInterno(R[i], j, S, lista_de_pares);
          i := i + 1;
       Senão
          LoopInterno(S[j], i, R, lista_de_pares);
          j := j + 1;
       FimSe
    FimEnquanto
Fim
```

O procedimento LoopInterno é definido como segue:

```
LoopInterno(t :  elemento; começo:  inteiro; R: nó;
            lista_de_pares:  lista de ((id, MBR),(id, MBR)))

Var k;

    k := começo;
    Enquanto (k ≤ || R ||) ∧ (R[k].x_min ≤ t.x_min) Faça // Interseção em X
       Se (t.y_min < R[k].u_max) ∧ (t.y_max > R[k].y_min) Então // Interseção em Y
          Adicione((t, R[k]), lista_de_pares);
       FimSe
       k := k + 1;
    FimEnquanto
Fim
```

A incorporação do *plane-sweeping* ao algoritmo de junção pode ser vista logo abaixo:

```
Junção_em_profundidade(R, S: nó; Intr:  MBR)
   // Intr é o MBR correspondente à interseção de R e S

   Elimine todas as entradas r ∈ R | r.MBR ∩ Intr = ∅;
   Elimine todas as entradas s ∈ S | s.MBR ∩ Intr = ∅;
```

```
   Classifique(R);
   Classifique(S);
   TesteInterseção(R, S, Lista_de_pares);
   Para (∀ (E_R,E_S) ∈ Lista_de_pares) Faça
      Se (E_R) é página) Então
         Inclua (E_R, E_S) no conjunto-resposta;
      Senão
         Leia(E_R.ponteiro);
         Leia(E_S.ponteiro);
         Junção_em_profundidade(E_R.ponteiro,E_S.ponteiro,E_R.MBR ∩ E_S.MBR);
      FimSe
   FimPara
Fim
```

Outro melhoramento incorporado é a fixação de páginas no *bufferpool* (*buffer pinning*). O *buffer pinning* consiste em fixar no *bufferpool* a página que possui o maior número de interseções (grau de interseção). Assim, após calculada a lista de pares de elementos que possuem interseção, daqui por diante denominada *índice intermediário de junção* (IIJ), procura-se determinar dentre estes elementos aquele que possui o maior grau. Ao efetuar uma leitura deste elemento, ele é fixado no *bufferpool* e só é eliminado quando todas as referências a ele são processadas.

O *buffer pinning* impede que uma página muito solicitada seja eliminada do *bufferpool* entre chamadas recursivas, por exemplo. Entretanto, aliado ao *buffer pinning* deve-se haver algum tipo de ordenação. A ordenação serve para que duas ou mais ocorrências de uma mesma página no IIJ não estejam muito distantes entre si no IIJ. Esta distância faz com que páginas que se encontram no *bufferpool* tenham probabilidade menor de serem descartadas antes que todas as referências a elas sejam processadas. O *buffer pinning* serve para garantir que a página mais referenciada não seja descartada. Além disso, a ordenação faz com que todas as ocorrências da página com maior grau estejam juntas, sendo processadas logo, e liberando o *bufferpool* para que outra página seja fixada em seu lugar.

Dentre as ordenações possíveis, podemos destacar a ordenação obtida pela rotina de *plane-sweeping*, a ordenação através de uma curva $z$, ou até mesmo percorrer o IIJ buscando referências desta página.

Este último algoritmo é o segundo algoritmo implementado e incorpora os melhoramentos de *buffer pinning*, redução do espaço de busca e ordenação discutidos.

O terceiro e último algoritmo implementado é o apresentado por Huang *et alii* em [HJR97]. Este algoritmo incorpora todos os melhoramentos propostos por Brinkhoff, mas propõe um percurso em largura, como apresentado no algoritmo abaixo:

```
Junção_em_largura(R, S: nó)
// hR = hS, altura das árvores;
Var Lista[hR]: lista de IIJ; i :  inteiro;

    Lista[hR] := ∅;
    i := 1;
    Lista[i] := junção(R, S, R.MBR ∩ S.MBR);
    Enquanto i ≤ hR Faça
       Para (∀ (E_R,E_S) ∈ Lista[i]) Faça
          Leia(E_R.ponteiro);
          Leia(E_S.ponteiro);
          Lista[i + 1] := junção(E_R.ponteiro, E_S.ponteiro, E_R.MBR ∩ E_S.MBR);
       FimPara
       i := i + 1;
    FimEnquanto
    Conjunto-resposta := Lista[i];
Fim
```

A função junção incorpora os melhoramentos propostos por Brinkhoff e pode ser definida como segue:

```
Função junção(R, S: nó, Intr :  MBR) : IIJ
    Elimine todas as entradas r ∈ R | r.MBR ∩ Intr = ∅;
    Elimine todas as entradas s ∈ S | s.MBR ∩ Intr = ∅;
    Classifique(R);
    Classifique(S);
    TesteInterseção(R, S, Lista_de_pares);
    Retorne(Lista_de_pares);
Fim
```

O percurso em largura é justificado pela possibilidade de se aplicar as otimizações inteiramente em relação a determinado nível. Como exemplo, considere páginas que seriam descartadas devido às recursões do percurso em profundidade. No percurso em largura é possível uma ordenação tal que as páginas com maior grau seriam lidas uma quantidade mínima de vezes. Assim, as otimização que seriam aplicadas localmente na junção de dois nós apenas, podem ser aplicadas a todos os nós de determinado nível.

Dentre as otimizações que podem ser aplicadas globalmente, destacamos o *buffer pinning* e ordenação dos IIJs. Entretanto, os critérios de ordenação dos IIJs propostos são diferentes dos propostos por Brinkhoff:

- sem ordenação, onde na realidade lê-se os pares na mesma ordem em que foram gerados pela rotina de *plane-sweeping*;
- ordenação em relação à $x_{min}$ dos elementos de uma das árvores;
- ordenação pela soma dos centros $x$ dos MBRs do par;
- ordenação pelo centro $x$ do MBR envolvente do par;
- ordenação pelo valor na curva de Hilbert relativo ao centro do MBR envolvente do par.

Todos os algoritmos de ordenação foram implementados e armazenados em uma biblioteca auxiliar, incluindo aqueles propostos em Brinkhoff. As rotinas de a manipulação de elementos necessários à escolha de páginas a serem fixadas no *bufferpool* também estão contidas em uma biblioteca auxiliar.

## 4.4 Os dados

Os conjuntos de dados utilizados para a construção dos índices são dados reais, obtidos junto ao projeto SAGRE (Sistema Automatizado de Gerência de Rede Externa) [Mag97, Mag93, MGS+94, Mag94], um sistema que utiliza dados geográficos, desenvolvido pelo Centro de Pesquisa e Desenvolvimento (CPqD), antigo CPqD-Telebrás. A função deste sistema é automatizar os processos relacionados ao cadastro, planejamento, projeto, implantação, manutenção, dentre outras operações, da rede externa das operadoras de telefonia. O termo *rede externa* se refere a toda a rede de telecomunicações que se encontra fora das estações telefônicas, o que inclui a rede de canalização (conjunto de dutos enterrados conectados a caixas subterrâneas), a rede aérea (rede de cabos suspensos) e a rede subterrânea (rede de cabos que passam pela rede de canalização).

Além da rede externa, o sistema mantém informações sobre o mapa urbano básico (formado por elementos como ruas, quadras, lotes, acidentes geográficos, obras públicas, etc), as estações telefônicas e os limites de gerência de rede, que são as linhas que delimitam as áreas de abrangência de estações, distritos, etc..

Os dados se referem à cidade de Valinhos, localizada a 7 km de Campinas, no estado de São Paulo. A utilização desses dados deve-se ao fato de manter a padronização com pesquisas anteriores [Car98], além de serem representativos em aplicações de utilidade pública, como telefonia, energia, água e esgoto, televisão à cabo. Os dados, entretanto, não foram utilizados em seu formato original. Para serem utilizados, tiveram que passar por uma filtragem. Os passos e a justificativa desta filtragem são explicados em [Car98].

No formato binário, constituem-se basicamente de uma seqüência de elementos do tipo *double float*. A cada quatro *doubles* temos um MBR da forma $(x_{min}, y_{min}, x_{max}, y_{max})$. As coordenadas mínimas e máximas encontradas nos dados são as seguintes:

- $x_{min}$ = 286.586,042
- $x_{max}$ = 307.716,81
- $y_{min}$ = 7.448.789,53
- $y_{max}$ = 7.465.892,47

Estes dados estão dispostos em três arquivos: o primeiro contendo pontos, denominado *postes* (Figura 4.3), o segundo composto por retângulos, denominado *quadras* (Figura 4.2), e um terceiro conjunto contendo pontos e retângulos, denominado *cidade* (Figura 4.4). O conjunto *postes* é composto de 13.813 MBRs, o conjunto *quadras* contém 2.473 MBRs e o conjunto *cidade* possui 66.837 MBRs, que engloba o conjunto de *postes*, o de *quadras* e outros elementos como caixas de telefone, tubos, cabos, área de ambrangência de estações telefônicas, dentre outros.

Pode-se notar que os dados têm uma distribuição não uniforme e tamanhos bastante variáveis. Estes MBRs foram indexados em três R*-trees distintas, uma de postes (pontos), outra de quadras (retângulos) e outra correspondendo ao conjunto *cidade*. As árvores foram montadas com tamanhos de página de $1k$, $2k$, $4k$ e $8k$. Na Tabela 4.1 temos uma relação das características das árvores com relação à ocupação média e altura, dentre outros fatores. Os resultados mostrados neste trabalho refletem execução sobre o conjunto *postes* e *quadras*, exceto quando outros índices forem explicitamente mencionadas.

## 4.5 Consultas testadas

As consultas foram montadas de forma a executar a junção entre o índice de postes e o índice de quadras, com páginas variando entre $1k$, $2k$, $4k$ e $8k$. O tamanho do *bufferpool* variou entre 2, 4, 8, 16, 24 e 32 páginas, inicialmente com e depois sem *buffer pinning*. Além do *buffer pinning*, testamos também as seguintes políticas de substituição de páginas: LRU (*least recently used*), LFU (*least frequently used*), SC (*second chance*) e aleatória (*random - RAN*).

A essas configurações, somam-se os diferentes critérios de ordenação propostos nos diferentes trabalhos. Assim, no algoritmo de junção em profundidade, testamos a ordenação segundo curva $z$ (SZ) e a ordenação obtida pela rotina de *plane-sweeping* (NS). No algoritmo de percurso em largura, utilizamos os seguintes critérios de ordenação: sem ordenação (NS), pela soma dos centros $x$ (SS), pelo centro $x$ do MBR envolvente do par

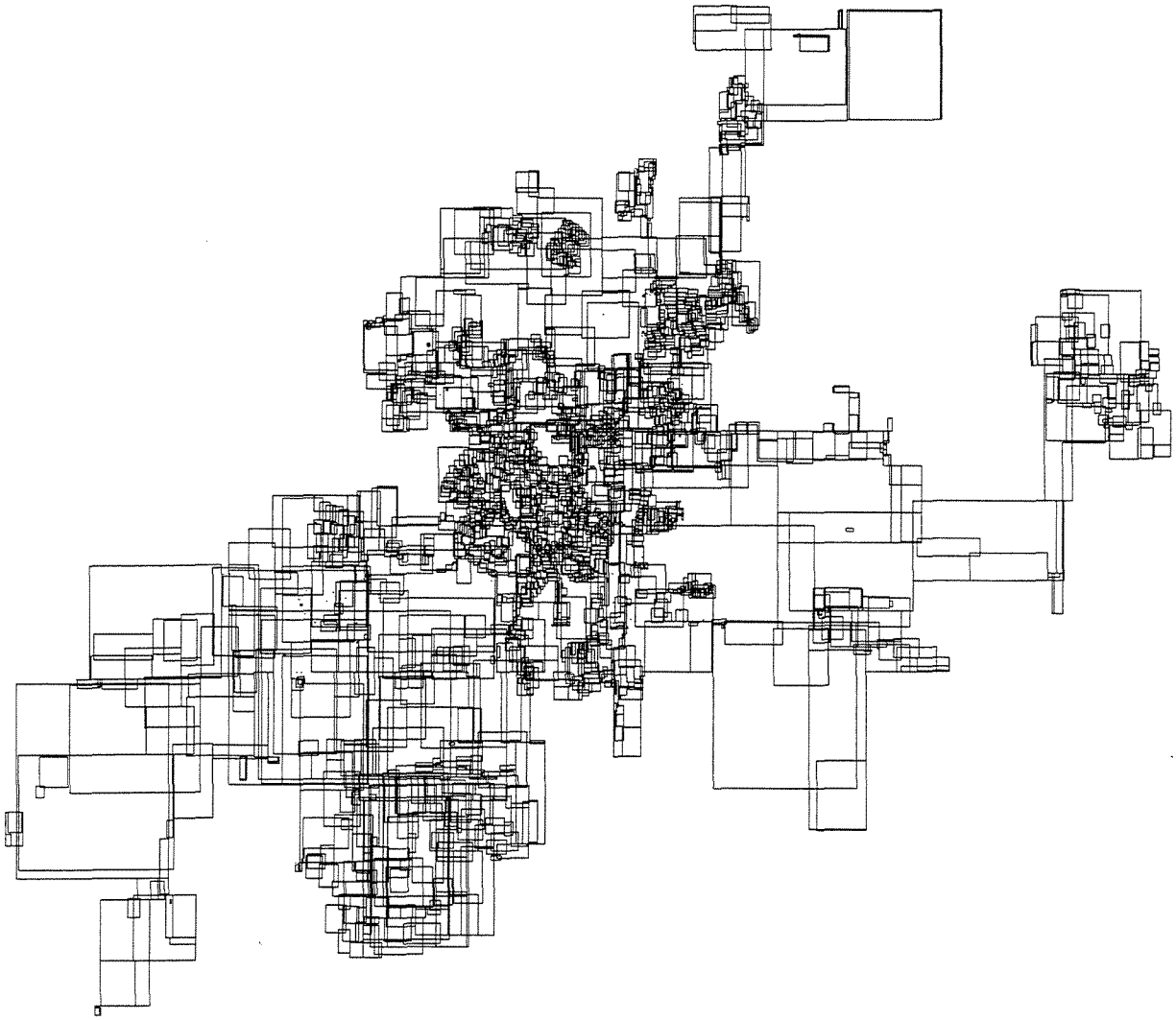


Figura 4.2: MBRs referentes ao conjunto de quadras da cidade de Valinhos.

Figura 4.3: Pontos referentes ao conjunto de postes da cidade de Valinhos.

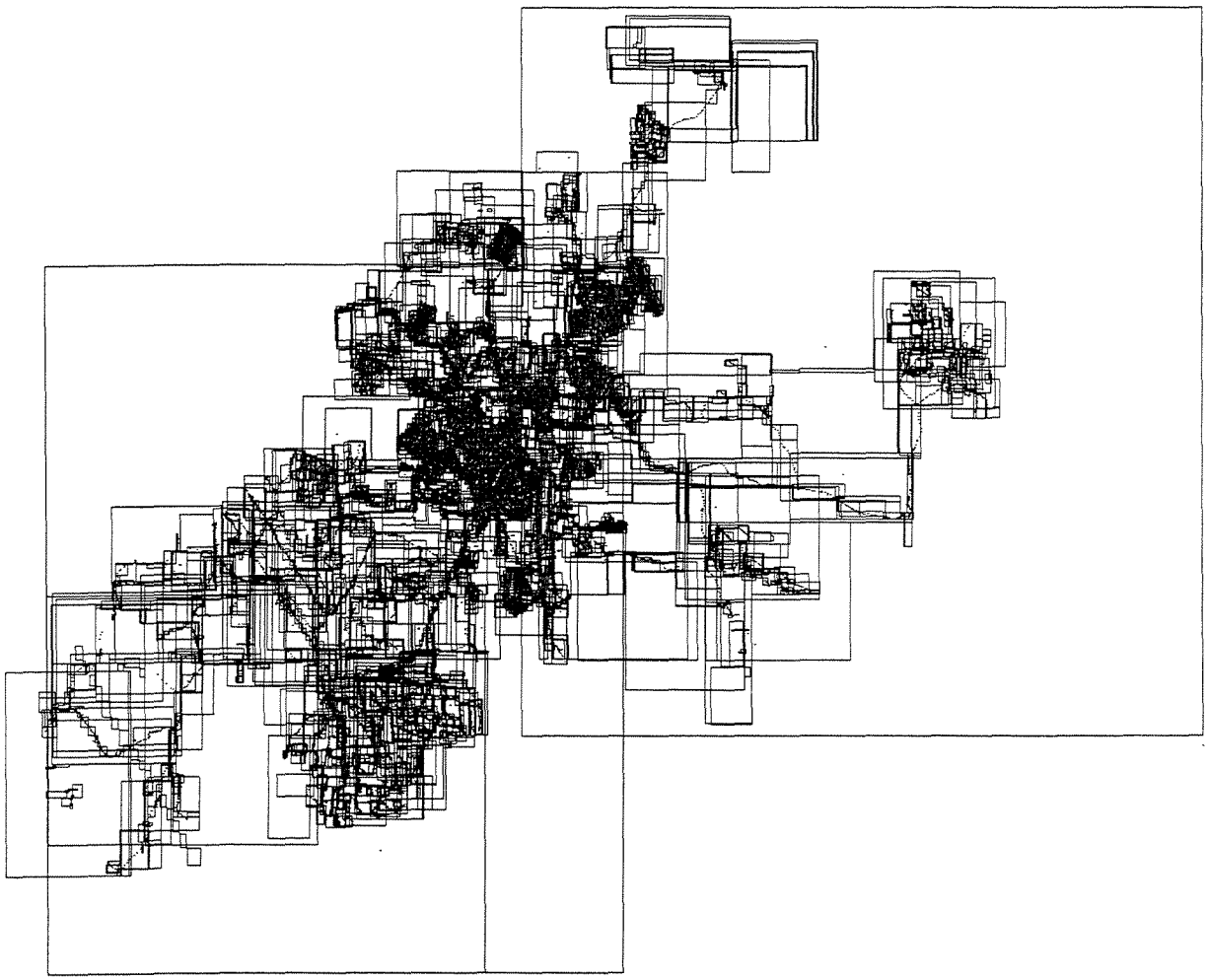


Figura 4.4: MBRs referentes ao conjunto *cidade*.

| Árvore | Tamanho da Página | Páginas Ocupadas | Altura | Número total de entradas | Ocupação Média |
|---|---|---|---|---|---|
| Postes | 1k | 870 | 3 | 14.682 | 67,5 % |
| Quadras | | 161 | 2 | 2.633 | 65,4 % |
| Cidade | | 4270 | 3 | 71.106 | 66,6 % |
| Postes | 2k | 420 | 2 | 14.232 | 66,4 % |
| Quadras | | 76 | 2 | 2.548 | 65,7 % |
| Cidade | | 2012 | 3 | 68.848 | 67,1 % |
| Postes | 4k | 209 | 2 | 14.021 | 65,7 % |
| Quadras | | 37 | 1 | 2.473 | 66,4 % |
| Cidade | | 974 | 2 | 67.810 | 68.3 % |
| Postes | 8k | 100 | 1 | 13.912 | 68,1 % |
| Quadras | | 20 | 1 | 2.473 | 61,0 % |
| Cidade | | 480 | 2 | 67.316 | 68.7 % |

Tabela 4.1: Características dos índices gerados, segundo seu conjunto de origem.

(SC), pela coordenada $x_{min}$ de um dos participantes (SO), pela coordenada de Hilbert do centro do MBR envolvente do par (SH).

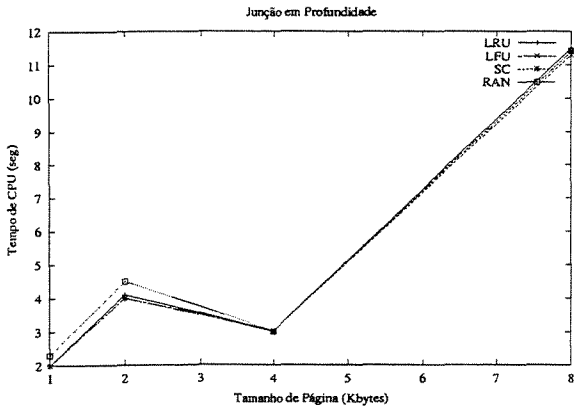
Cada consulta foi executada 10 vezes, computando o tempo de CPU de cada execução e tomando-se a média destes tempos. Para manter coerência entre uma execução e outra, todas as execuções foram efetuadas a partir das mesmas condições iniciais.

## 4.6 Resultados obtidos

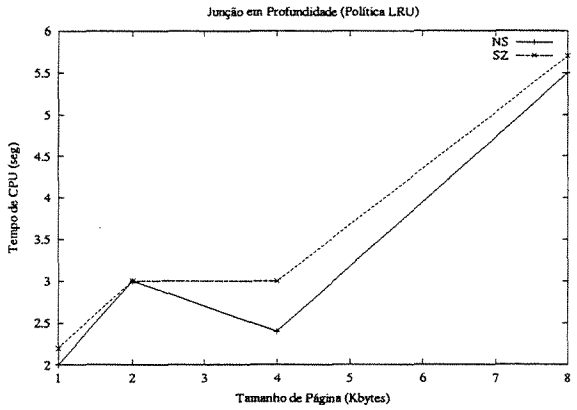
Apresentamos agora os resultados obtidos através das experiências realizadas. Dividimos os resultados em duas partes principais: primeiramente os resultados das medições de tempo de CPU, e subseqüentemente, o resultados segundo as operações de entrada/saída.

### 4.6.1 Tempo de CPU e utilização de memória principal

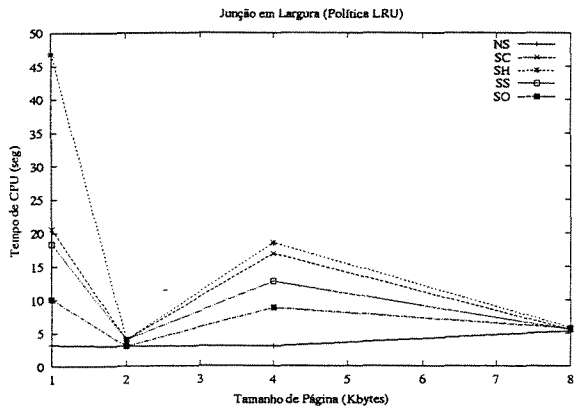
As medições efetuadas com relação ao tempo de CPU não são o alvo principal de nosso trabalho. Entretanto, algumas medições foram feitas de forma que pudéssemos ter uma estimativa da influência dos critérios de ordenação, políticas de substituição de páginas e o tamanho das páginas de disco no desempenho dos métodos. Os resultados apresentados refletem a média referente a 10 execuções, utilizando *buffer pool* com 32 páginas, embora o tempo não tenha variado significativamente para *buffers* menores.



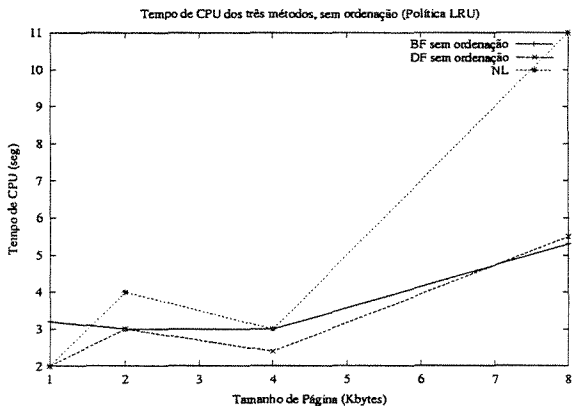

(a) Método NL




(b) Método DF com LRU




(c) Método BF com LRU




(d) Comparação entre NL, DF e BF com LRU

Figura 4.5: Tempo de CPU gasto na operação de junção segundo os algoritmos NL (a), DF (b) e BF (c). Os resultados dos métodos DF e BF estão reportados com os resultados obtidos com a política LRU. Em (d) temos uma comparação entre os três métodos, utilizando a política LRU, sem considerar os tempos necessários à classificação dos IIJs.

Na Figura 4.5 (a) temos o tempo de CPU[1] gasto com o método NL. Podemos observar que o tempo de CPU neste caso é tremendamente influenciado pelo tamanho da página utilizado na construção do índice. Isto se deve ao fato de que, com páginas menores, a árvore é mais alta e os nós envolvem uma área menor, contendo menos elementos. Isto possibilita uma maior capacidade de filtragem dos níveis interiores da árvore, havendo menos comparações e, conseqüentemente, menos tempo de CPU é necessário.

Nota-se também que a curva correspondente ao tempo de CPU do método NL faz uma sela em relação aos índices montados com páginas de $2k$ e $4k$: mesmo com página maior, o índice de $4k$ possui desempenho melhor em relação ao tempo de CPU do que os índices com $2k$. Isto se deve à diferença de altura entre os índices. As árvores de $2k$ possuem alturas iguais, enquanto que as árvores com páginas de $4k$ possuem alturas diferentes, com o índice correspondente às quadras possuindo um nível a menos que o de postes. Quando dois índices possuem alturas diferentes, o algoritmo inicia uma seqüência de *range queries* utilizando os MBRs do nó folha do índice mais baixo sobre a sub-árvore pertencente à árvore mais alta. Ou seja, no próximo nível, não estaremos mais comparando dois nós em *loops* aninhados, mas comparando todos os elementos de um nó com uma só janela de consulta: menos comparações. Neste caso específico, o fato de haver um nível a menos em um dos índices conseguiu contrabalançar o aumento de página, fazendo com que o aumento de página não se refletisse em aumento de CPU.

Esta vantagem nem sempre pode ser obtida. Isto pode ser verificado no gráfico da Figura 4.6, que mostra o desempenho da operação de junção NL sobre índices montados utilizando o conjunto *postes* e o conjunto *cidade* (veja seção 4.4), utilizados como contra-prova. Os índices gerados possuem as características descritas na Tabela 4.1.

Entre os índices de $1k$ e $2k$ o aumento é pequeno, mas evidencia a influência do tamanho da página. Entretanto, ao utilizar páginas de $4k$, há um aumento muito acentuado no tempo de CPU. Note que devido ao aumento de $1k$ para $2k$, saímos de uma junção entre duas árvores de alturas iguais para uma junção entre árvores de altura diferente. O fato de utilizar *range queries* diminuiu o impacto do aumento de página. Mas com $4k$, novamente lidamos com árvores de alturas iguais. Estas mesmas observações podem ser feitas para quando aumentamos a página para $8k$, quando novamente lidamos com árvores de alturas diferentes: o aumento não foi tão acentuado.

Os métodos de substituição de páginas praticamente não tiveram influência no tempo de CPU dos métodos NL, DF e BF, exceto pelo método RAN, que teve desempenho um pouco inferior. Devido a este fator, reportaremos o desempenho dos métodos DF e BF segundo a política LRU.

No gráfico da Figura 4.5 (b) temos o desempenho do método DF em relação ao tempo

[1] O tempo de CPU inclui apenas o tempo gasto nas operações que exigem processamentos, como exemplo operações aritméticas, operações de comparação, dentre outras.

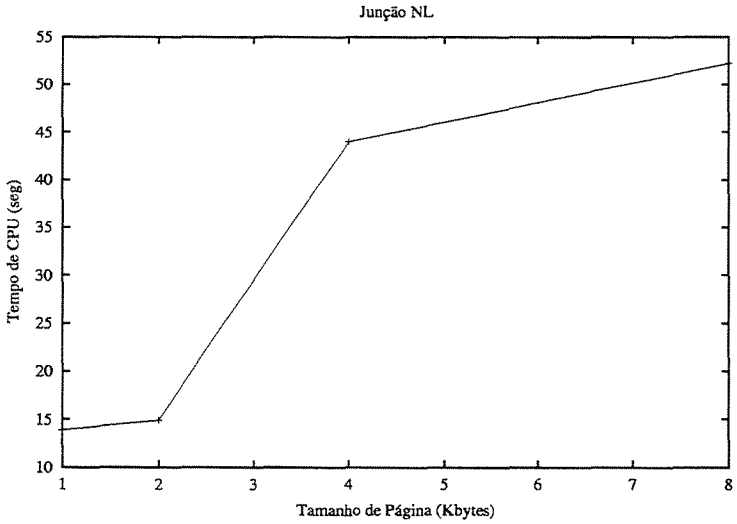



Figura 4.6: Tempo de CPU segundo o algoritmo NL na execução de junção sobre os índices *cidade* e *postes*.

de CPU, utilizando a política LRU. Note que, também o DF sofre influência do tamanho da página, da mesma forma que o método NL. Entretanto, a redução do espaço de busca e a utilização da técnica de *plane-sweeping* fazem com que o tempo de CPU do método DF seja menor do que o do método NL. Pode-se observar também que o método de classificação segundo a ordem $z$ (SZ) teve tempo de CPU ligeiramente superior ao método sem ordenação (NS). Embora denominado *sem ordenação*, na realidade há uma certa organização, pois este critério recebe um IIJ com grupos ordenados pela coordenas $x_{min}$ de um dos elementos do par. Esta ordenação é obtida através da rotina de *plane-sweeping*. Estes tempos poderiam ser ainda melhores se alguns ajustes finos fossem implementados, aumentando ainda mais a vantagem em relação ao método NL.

Na Figura 4.5 (c) podemos observar os tempos de CPU relacionados ao método de junção em largura (BF). Podemos observar que a junção utilizando curva de Hilbert (SH) é a mais custosa em termos de CPU. E realmente o cálculo dos valores de Hilbert é bastante caro, em torno $O(n^2)$. Seguem empatadas a ordenação pela soma dos centros (SS) e a ordenação pelo centro do MBR envolvente do par (SC), seguidas da ordenação segundo a coordenada $x_{min}$ de um dos elementos do par (SO). É fácil perceber que o tempo de CPU é extremamente baixo para quando não há ordenação. Novamente, no método *sem ordenação* há uma certa organização local, gerada pela rotina de *plane-sweeping*.

Para o método BF, pode-se notar que há uma diferença muito grande entre as páginas de $1k$ e $4k$, com alto custo de CPU, e $2k$ e $8k$, com custo mais baixo. Uma explicação para

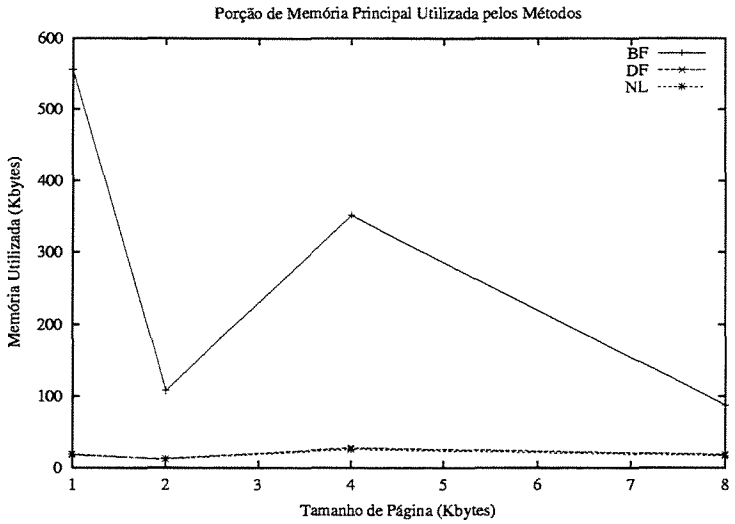



Figura 4.7: Utilização de memória RAM segundo os métodos *nested-loop* (NL), junção em profundidade (DF) e junção em largura (BF).

este fator pode ser extraída da Figura 4.7, que mostra a utilização de memória principal dos métodos. Observamos que os métodos NL e DF utilizam muito pouca memória, comparados ao método BF. Isto se deve ao fato de que o percurso utilizado nos primeiros métodos cobre apenas dois nós em cada nível da árvore. Já o método BF executa a junção em todos os nós de um dado nível antes de descer ao próximo nível da árvore. Desta forma, mais memória é necessário para armazenar os índices intermediários de junção.

Portanto, a maior utilização de memória em $1k$ e $2k$ é devido a um número maior de interseções computadas em níveis internos das estruturas, ao mesmo tempo. Havendo mais pares de elementos formando o IIJ, o esforço de ordenação é maior, pois mais operações aritméticas (dentre outras) são necessárias.

Uma outra observação pode ser feita com relação à influência do tamanho da página em relação ao tempo de CPU. O método BF também tem seu tempo de CPU afetado pelo aumento da página. Isto pode ser observado nos gráficos em relação à curva NS (classificações locais obtidas através da técnica de *plane-sweeping*), que tem um pequeno aumento conforme a página aumenta.

Esta relação pode ser melhor visualizada no gráfico da Figura 4.5 (d), que compara o tempo de CPU segundo a política LRU sem aplicar técnicas de classificação nos métodos DF e BF. Neste gráfico podemos ver como os melhoramentos (redução do espaço de busca e *plane-sweeping*) influenciam na redução do tempo de CPU. Entretanto, o impacto do tempo de CPU nas medições feitas com o método DF e BF está sobrestimado, uma vez

que não nos preocupamos em efetuar um ajuste fino neste critério, pois considera-se como alvo de nosso trabalho o número de operações de entrada e saída. Mesmo o tempo da técnica NS do método BF sofreria uma redução maior com melhores ajustes.

Assim, os picos encontrados na curva correspondente ao método BF com páginas de $1k$ e $4k$, caso fosse feito um ajuste fino, poderiam ser menores do que os mostrados aqui. O custo maior de CPU nestes casos estão diretamente relacionados com o tamanhos dos índices intermediários de junção. Entretanto, embora esteja claro que o método BF utilize muito mais memória do que os método NL e DF, a quantidade exata dependerá da *densidade* dos dados, se considerarmos a junção por interseção.

Pode-se notar também uma diferença entre os tempos de CPU do método BF e DF em relação a páginas de tamanho $1k$, $2k$, $4k$. Esta diferença pode ser explicada pelo esforço adicional na manipulação dos IIJs (inserções e remoções), bem maiores no método BF do que no método DF. Como já foi mencionado, não nos preocupamos em fazer um ajuste fino em relação ao tempo de CPU, embora as relações de causa/efeito estejam conservadas.

### 4.6.2 Operações de entrada/saída

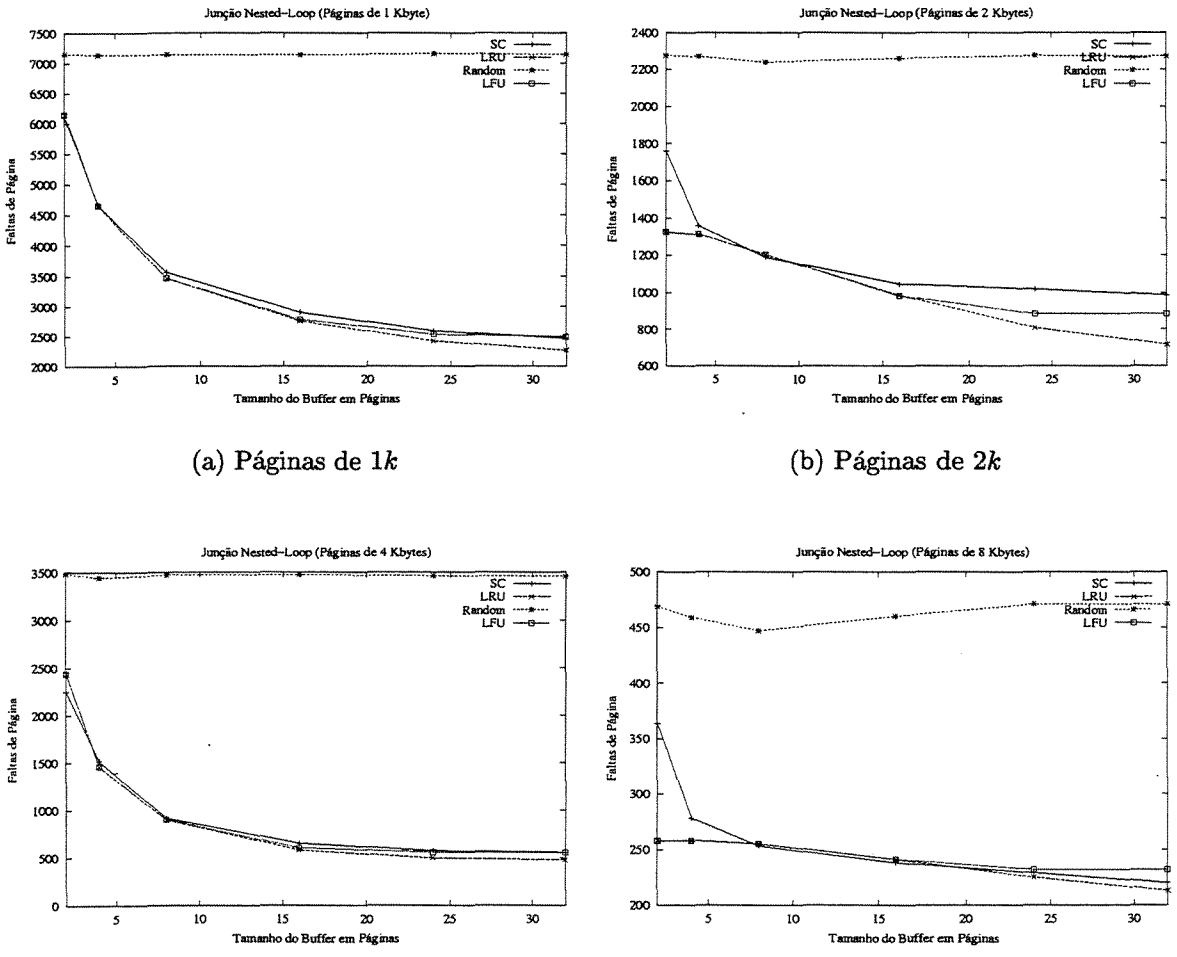
Nesta seção, veremos os resultados obtidos pelos métodos em relação às operações de entrada e saída. Inicialmente, mostraremos os resultados obtidos com o método *nested-loop* (NL) e em seguida resultados do método de junção em profundidade (DF) e do método de junção em largura (BF). Finalmente, mostraremos uma análise comparativa entre os métodos, para verificar os ganhos obtidos em relação aos melhoramentos propostos pelos autores e a influência que o tamanho de página e o tamanho do *bufferpool* têm no desempenho destes métodos.

#### Resultados obtidos com o método NL

Nos gráficos mostrados na Figura 4.8 temos o desempenho obtido pelo método NL, divididos em relação ao tamanho da página utilizada na indexação. Em cada gráfico, temos as curvas resultantes das diferentes políticas de substituição de página.

Pelos gráficos, é possível ver que o melhor desempenho foi obtido utilizando a política *least recently used* (LRU), seguida da *least frequently used* (LFU), a *second chance* (SC) e a *random*. A política de substituição aleatória (*random*) é a que não conseguiu tomar vantagem do aumento no tamanho do *buffer pool*. Uma vez que o melhor desempenho foi obtido utilizando a política LRU, utilizaremos as curvas relacionadas a esta política nas comparações que serão feitas com os métodos DF e BF.

A surpresa desta medição é o desempenho do método LFU com páginas de $8k$, que ficou abaixo do desempenho da política SC para *buffers* maiores que 8 páginas. Isto é




(a) Páginas de $1k$

(b) Páginas de $2k$

(c) Páginas de $4k$

(d) Páginas de $8k$

Figura 4.8: Número de operações de entrada/saída segundo o método NL, com índices montados em páginas de $1k$ (a), $2k$ (b), $4k$ (c) e $8k$ (d).

devido à própria política de substituição de páginas e a determinadas seqüências de leitura de página. Para entender este problema, é necessário saber como funciona o LFU.

O LFU é uma política de substituição de páginas que mantém um contador de freqüência para cada página do *bufferpool*. Assim, se o *bufferpool* contém oito páginas, há oito contadores de páginas. Inicialmente, todos os contadores estão com freqüência zero. A cada leitura/escrita de página, a política LFU executa *shifts* para direita em todos os contadores de página do *bufferpool*. Isto assegura que as páginas que não foram referenciadas tenham suas freqüências diminuídas, mas mantenham sua ordem de freqüência. A página recém referenciada recebe como freqüência sua freqüência anterior (já "shiftada") mais um valor cujo bit mais significativo é igual a 1 e o restante igual a 0. Isto faz com que esta página seja a que tem maior freqüência agora, com as demais páginas mantendo a antiga relação de freqüência.

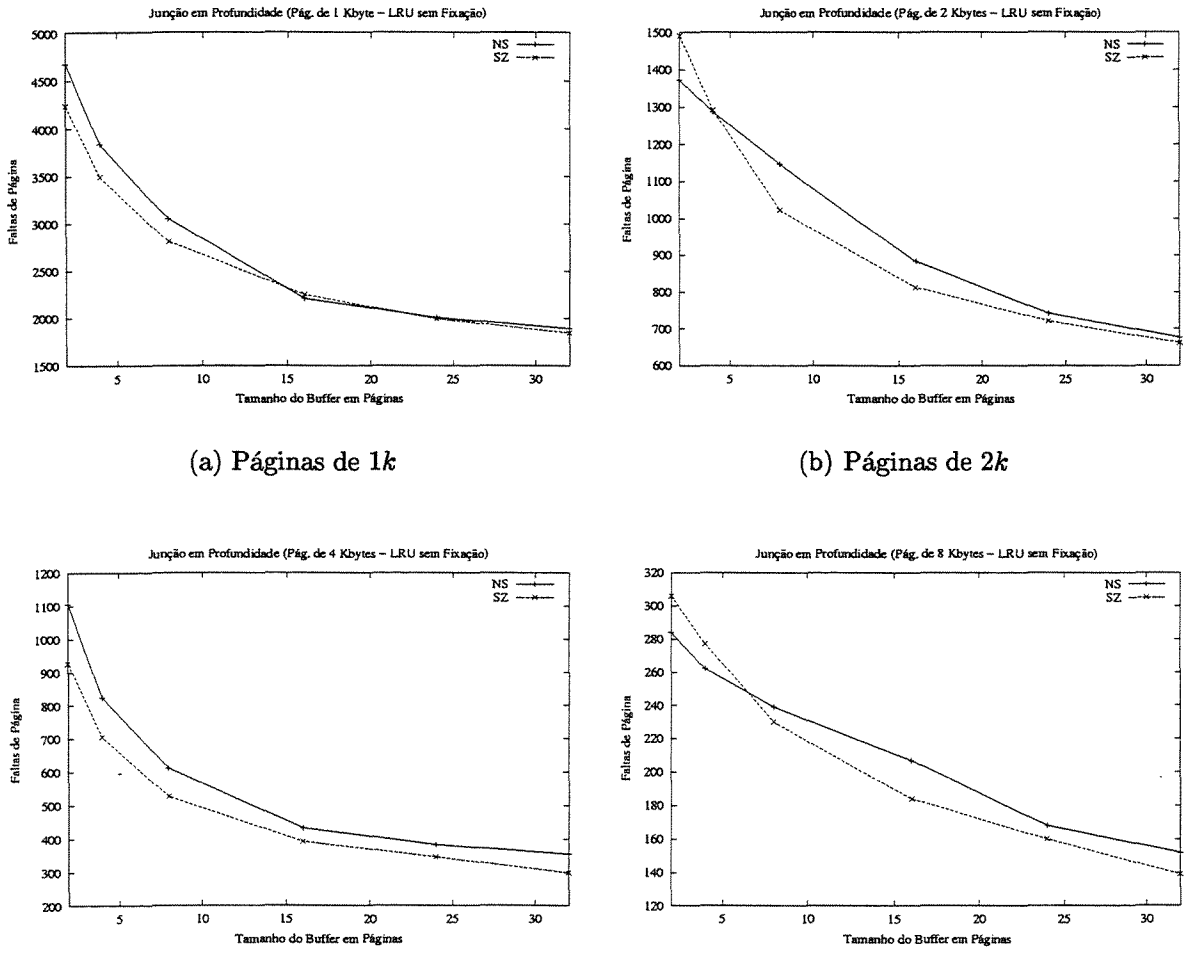
Na seqüencia de leituras/escritas, pode ocorrer que uma página tenha atingido a freqüência igual a zero, o que permite que seja substituída. Assim, a posição que ela ocupava pode ser utilizada por outra página, mesmo que haja uma posição do *bufferpool* que nunca tenha sido utilizada antes. Pode ocorrer que, após esta substituição, a página que acabou de ser descartada seja novamente requisitada, gerando um *miss*. É possível que haja uma seqüência de leituras de página em que ocorra fatos deste tipo em seqüência, o que prejudica o desempenho do método.

### Resultados obtidos através do método DF

Nos gráficos da Figura 4.9, apresentamos os resultados do método DF, utilizando a política LRU de substituição de páginas, sem fixação de páginas no *bufferpool*. Os resultados obtidos com as outras políticas de substituição, embora não mostrados aqui, são coerentes com os resultados do método NL: LRU é a melhor política, seguido de LFU, SC e *random*, nesta ordem.

Em 4.9 (a), temos a evolução do método DF aplicado sobre índices com tamanho de página de $1k$, inicialmente sem ordenação (NS) dos IIJs e depois com a ordenação por curva $z$ (SZ). Note que as medições em relação aos índices de $1k$ foram feitas sob condições drásticas de *bufferpool*. A razão do número de páginas do *bufferpool* sobre o número de páginas em disco é muito pequena. Em conseqüência disso, o desempenho do método é limitado pelo tamanho do *buffer*, de forma que nem o método NS nem o método SZ conseguem se destacar. Outros resultados, mostrados na Tabela 4.2 e obtidos com proporções maiores de *bufferpool*, mostram que a ordenação $z$ oferece melhor desempenho para o algoritmo de junção com índices montados em páginas de $1k$.

Estas mesmas observações podem ser feitas baseando-se nos gráficos das Figuras 4.9 (b), 4.9 (c) e 4.9 (d). Uma vez que a proporção de páginas no *bufferpool* aumenta em relação à quantidade de páginas presentes no disco, o desempenho da ordenação SZ




(a) Páginas de $1k$

(b) Páginas de $2k$

(c) Páginas de $4k$

(d) Páginas de $8k$

Figura 4.9: Resultados obtidos com o método DF, política LRU, executando sobre índices de $1k$ (a), $2k$ (b), $4k$ (c) e $8k$ (d).

| Tipo de Classificação | Tamanho do *Bufferpool* | Número de Acessos |
|---|---|---|
| sem ordenação | 64 | 1673 |
| ordenação $z$ | 64 | 1565 |
| sem ordenação | 128 | 1400 |
| ordenação $z$ | 128 | 1354 |

Tabela 4.2: Número de acessos obtidos com o método DF, utilizando índices de $1k$, com *bufferpool* maior e política LRU.

mantém uma boa vantagem em relação ao método NS.

Podemos nos perguntar se o tempo gasto na ordenação dos IIJs compensa o ganho obtido com as operações de entrada/saída (E/S). Neste aspecto, Brinkhoff [BKS93], na época em que propôs o método, disse que o ganho obtido com a ordenação não era compensador. Entretanto, a capacidade dos processadores atuais cresceu de forma espantosa, enquanto que a velocidade das unidades de disco não cresceu na mesma proporção.

Considerando este aspecto, podemos fazer uma análise rústica: considere que a operação de transferência do disco para memória leve em média 20 $ms$, levando em conta tempo de latência, tempo de posicionamento do braço, dentre outros fatores, independente do tamanho do bloco. Observamos que, para o índice montado com páginas de $4k$, a diferença entre a classificação NS e SZ é de aproximadamente 0,5 segundo. Ainda para página de $4k$, com *bufferpool* igual a 32 páginas, há uma diferença de aproximadamente 50 leituras de páginas a mais executadas pelo critério NS. Multiplicando $50 \times 20ms$ temos que o tempo gasto nessas 50 leituras a mais de páginas é igual a 1 segundo, mais do que o 0,5 segundo gasto na ordenação. Ou seja, gastamos 0,5 segundo para ordenar, mas economizamos 1 segundo em leituras de páginas, ganhando mais 0,5 segundo.

Considerando-se ainda que há várias otimizações que podem ser aplicadas de forma a melhorar o tempo de CPU, a diferença entre as curvas relativas à ordenação NS e SZ quanto ao tempo de CPU seria ainda menor. Desta forma é mais vantajosa a utilização da ordenação SZ do que a utilização da ordenação NS. O algoritmo DF fica então limitado apenas em relação às operações de E/S (entrada/saída). Poderíamos então aumentar indefinidamente o tamanho da página até que tenhamos que efetuar apenas uma operação de E/S? Sim, mas nesse caso a operação ficaria novamente limitada pelo tempo de CPU (veja como o tempo de CPU aumenta conforme a página aumenta). Além disso, teríamos os problemas advindos da busca, atualização, remoção, dentre outras operações e características indesejáveis de um arquivo com registros seqüenciais, com o agravante de estarmos lidando com dados espaciais. Ou seja, deve haver um compromisso entre

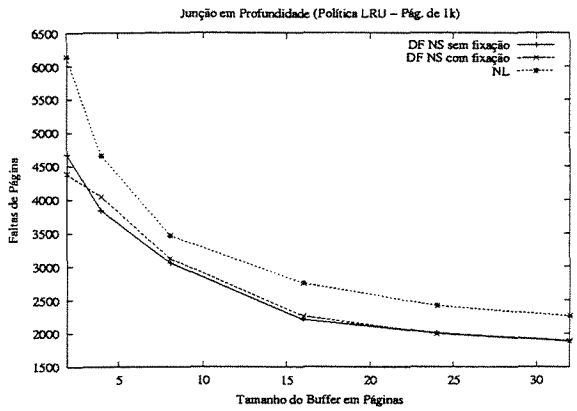
o tamanho da página e capacidade de processamento. É necessário mais e mais buscar algoritmos eficientes para a computação de predicados espaciais, conforme aumentamos o tamanho das páginas.

Passamos agora à análise do método DF com a fixação de páginas no *bufferpool*. A questão é determinar se a fixação de páginas no *bufferpool* vale a pena. Para responder esta pergunta, voltemo-nos aos gráficos da Figura 4.10, onde temos a comparação entre a classificação NS com e sem fixação e o método NL, e nos gráficos da Figura 4.11, que mostra a mesma comparação, mas em relação à classificação por curva $z$. Para *buffers* muito pequenos em relação à quantidade total de páginas em disco, a desvantagem do método com fixação é evidente: a fixação de páginas implica em menos páginas disponíveis no *bufferpool*, acarretando mais falhas de página no nível sendo processado no momento.

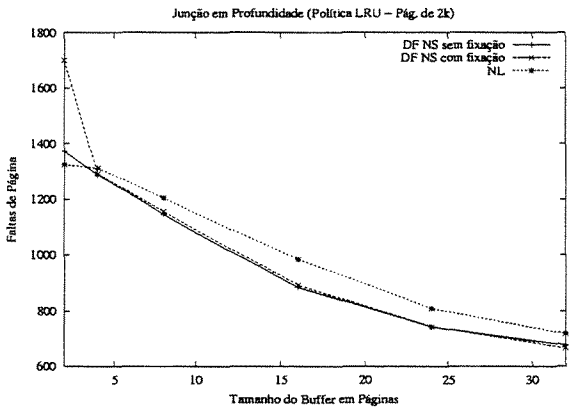
Entretanto, à medida que o *bufferpool* cresce, as técnicas com fixação começam a levar uma pequena vantagem. Embora pequena, esta vantagem ainda é compensadora, pois a carga de CPU necessária para o cálculo do grau de interseção dos MBRs e a manutenção da lista de graus é muito pequena. Quando o *bufferpool* atinge aproximadamente 10% da quantidade de páginas presentes no disco, os resultados obtidos utilizando e não utilizando fixação são praticamente idênticos. Havendo mais páginas, a probabilidade de que uma página de níveis anteriores seja descartada é menor. Além disso, a maior quantidade de páginas no *bufferpool* também favorece mais *hits* de páginas do nível sendo processado, o que pode compensar uma eventual substituição de páginas de níveis superiores do índice.

A Figura 4.12 apresenta os resultados obtidos com o método DF utilizando fixação de páginas no *bufferpool*, para os critérios de classificação SZ e NS, comparados como os resultados do método NL. Através deles é possível tirar as mesmas conclusões em relação ao DF sem fixação, apresentados nos gráficos da Figura 4.9.

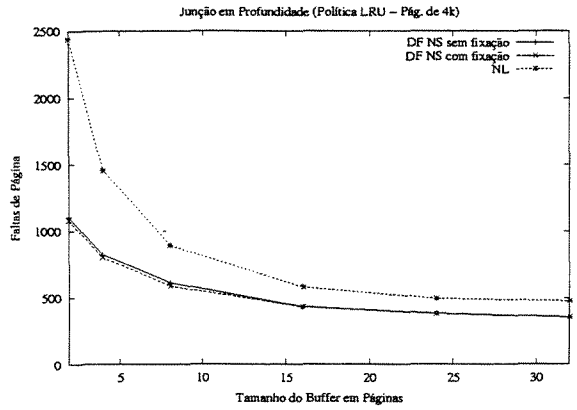
Dois outros experimentos resultaram em desempenho pior que o esperado: a operação de *purge*, ou seja, eliminação de uma página do *bufferpool* após seu grau atingir zero, e o rearranjo do IIJ, executado para que todos os pares de um IIJ que contém o elemento que deve ser fixado estejam na seqüência. O *purge* não resultou em melhor desempenho devido ao fato de que, mesmo que determinada página tenha atingido grau zero, ainda é possível que ela venha a ser necessária. Isto se deve à natureza do algoritmo DF: sendo executado em profundidade, não sabemos se determinada página será necessária novamente ao sairmos de uma recursão. O rearranjo não resultou em melhor desempenho devido ao fato que outras páginas também estão envolvidas no par e não apenas a que deverá ser fixada. Este rearranjo destrói a proximidade que existia anteriormente, gerada pelo algoritmo de *plane-sweeping*, fazendo com que as referências de uma mesma página estejam mais dispersas no IIJ. Os resultados reportados nos gráficos foram os obtidos sem rearranjo e sem *purge*.



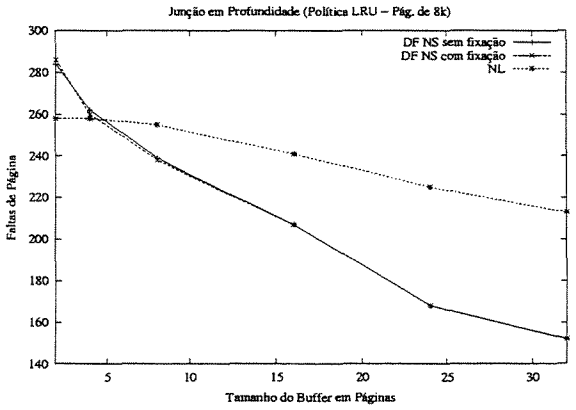

(a) Página de $1k$




(b) Página de $2k$

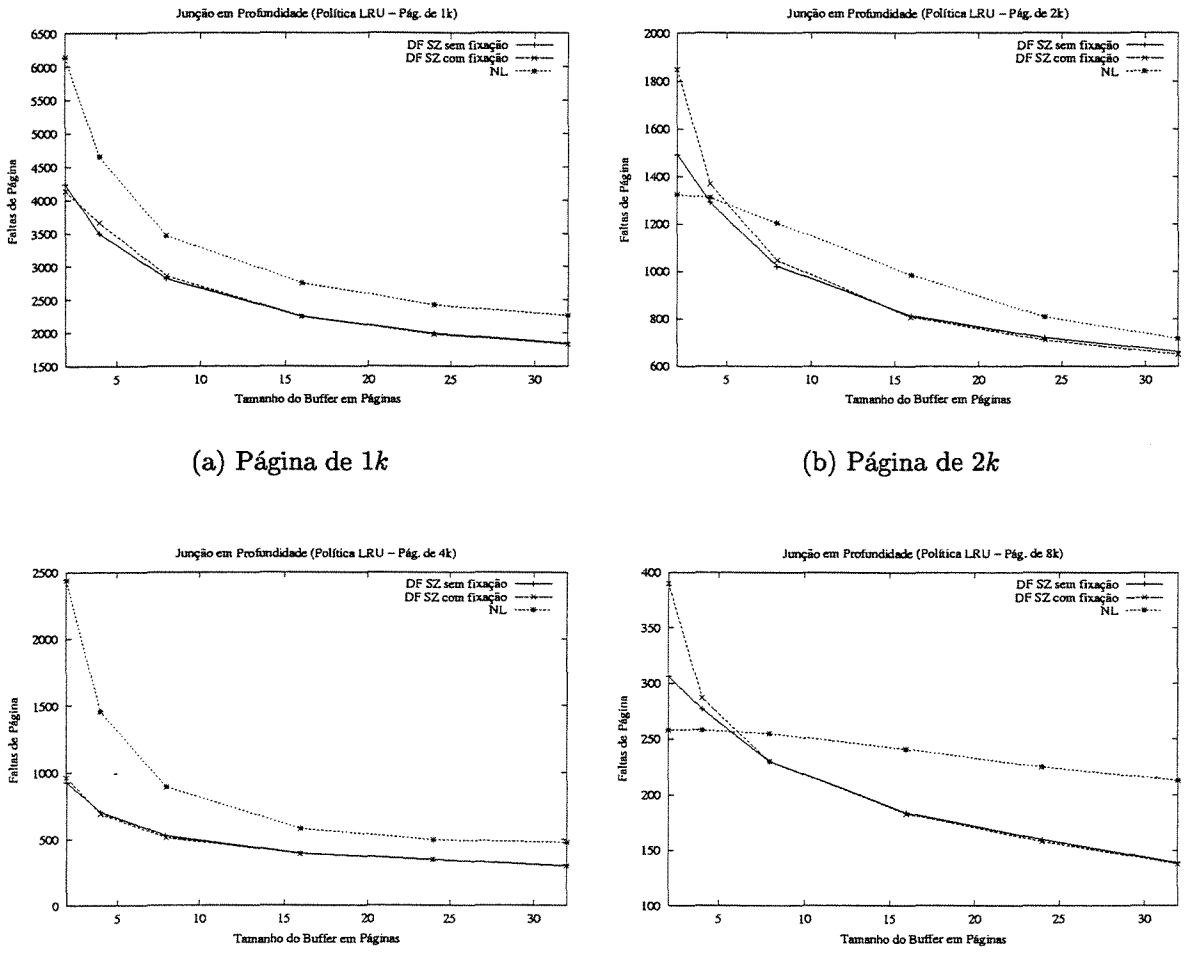



(c) Página de $4k$




(d) Página de $8k$

Figura 4.10: Comparação entre o método DF utilizando classificação NS, com fixação e sem fixação, em índices com página de $1k$ (a), $2k$ (b), $4k$ (c) e $8k$ (d).



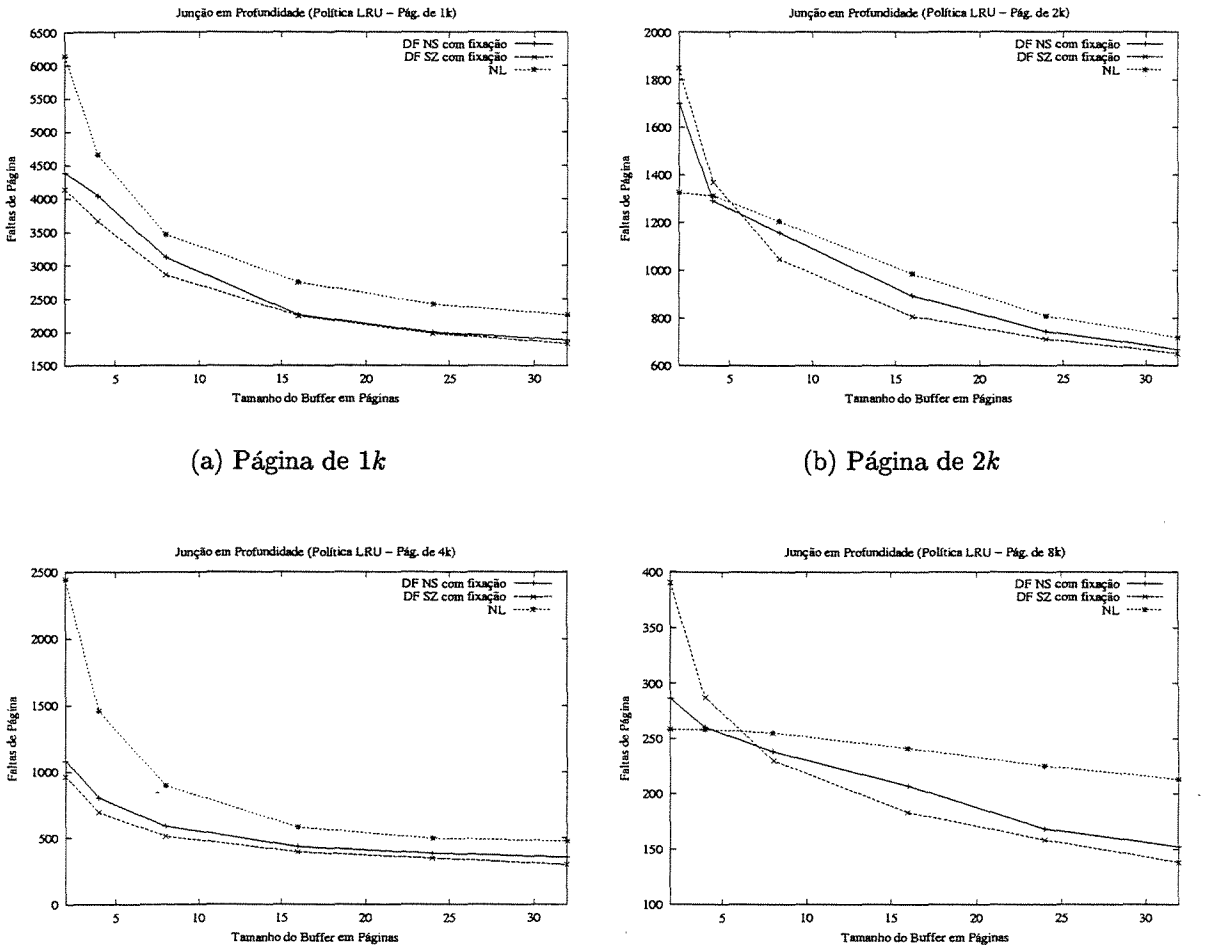

(a) Página de $1k$

(b) Página de $2k$

(c) Página de $4k$

(d) Página de $8k$

Figura 4.11: Comparação entre o método DF utilizando classificação SZ, com fixação e sem fixação, em índices com página de $1k$ (a), $2k$ (b), $4k$ (c) e $8k$ (d).




(a) Página de $1k$

(b) Página de $2k$

(c) Página de $4k$

(d) Página de $8k$

Figura 4.12: Número de operações de entrada/saída segundo o método DF, com fixação de páginas, sobre índice com página de $1k$ (a), $2k$ (b), $4k$ (c) e $8k$ (d), utilizando os critérios de ordenação: *sem ordenação* (NS) e *ordenação z* (SZ).

### Resultados obtidos através do método BF

Apresentamos agora o desempenho do método BF. Estes resultados são os obtidos com a política LRU. As medições feitas com as políticas LFU, SC e *random* refletiram os resultados dos outros métodos, ou seja, o método LRU foi o melhor, seguido do LFU, SC e *random*, nesta ordem.

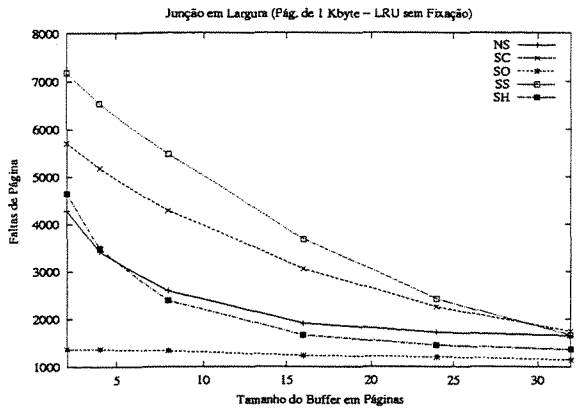
É facil de se notar que a ordenação SO é bem superior às demais quando as árvores possuem alturas diferentes. Isto permite a ordenação pelos elementos referentes à árvore mais alta, fazendo com que as referências de um mesmo nó estejam bem próximas no IIJ, ainda mais que os nós de determinada árvore não mais terão que concorrer com os nós da outra árvore. Quando os índices possuem alturas iguais, a ordenação SO é superior em quantidades mais generosas de *bufferpool*, mas não com tanto destaque.

As medições feitas com fixação de páginas no *bufferpool* podem ser vistas nos gráficos da Figura 4.14. Por eles é possível ver que o critério de ordenação SO também é superior, refletindo os resultados das execuções do BF sem fixação. Além disso, da mesma forma que o método DF, o BF com fixação também sofre quando o tamanho do *bufferpool* é muito pequeno, havendo aumento no número de *misses*. Mas, conforme o tamanho do *bufferpool* aumenta, as versões com fixação têm um pequeno ganho. Entretanto, como no método DF, conforme o *bufferpool* atinge aproximadamente 10% da quantidade de páginas em disco, as versões com e sem fixação ficam empatadas.

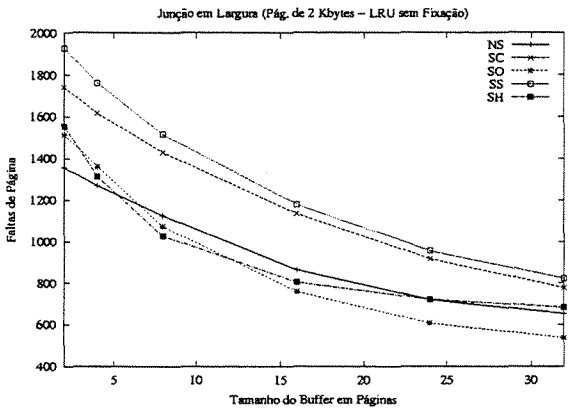
Ao contrário do método DF, o método BF foi capaz de executar apenas uma leitura de cada página existente no disco. Isto se deve a uma melhor distribuição das páginas nos IIJs, que no método BF atinge todas as páginas necessárias de determinado nível. Desta forma, sabe-se de antemão todas as páginas que serão necessárias para processar o próximo nível.

Em contrapartida, o método BF gastou muito mais memória para armazenar os IIJs e mais tempo de CPU para ordená-los, embora ainda haja espaço para melhora sob este aspecto em nossa implementação, do que o método DF. Mas o principal ponto contra este método continua sendo o tamanho dos IIJs. Dependendo do tamanho do espaço de execução alocado para o método, é possível que parte do IIJ deva ser armazenado em disco, o que acarretaria mais acessos a disco e que pode até mesmo anular as vantagens obtidas com o BF sobre o DF. Huang *et alii* [HJR97] fazem um estudo a este respeito em sua proposta. Para se ter uma idéia do impacto do tamanho dos IIJs, uma medição utilizando o conjunto *cidade* indexado com páginas de $1kbyte$ como os operandos da junção, utilizou uma porção de memória equivalente a $5,9Mbytes$, sendo que a soma dos tamanhos dos dois índices é aproximadamente $8,34Mbytes$.

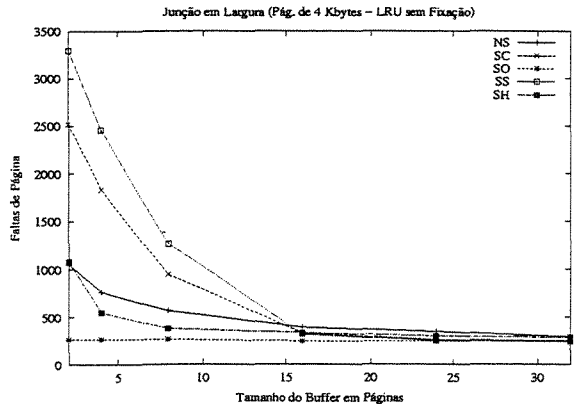
A comparação entre o método NL e as otimizações vencedoras entre os métodos DF e BF é apresentada nos gráficos da Figura 4.15 para páginas de *1 kbyte* (a), *2 kbytes* (b), *4 kbytes* (c) e *8 kbytes* (d). Por eles é possível ver que o método BF, se considerado apenas o



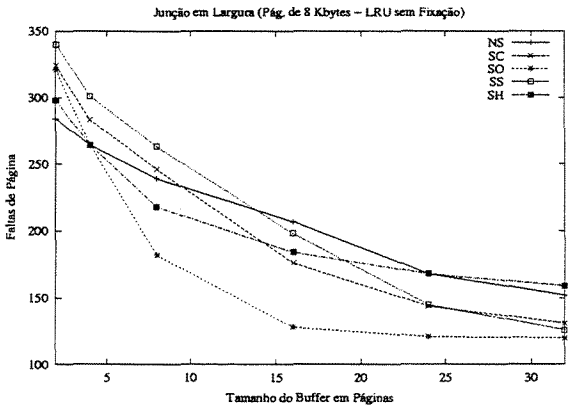

(a) Páginas de $1k$




(b) Páginas de $2k$

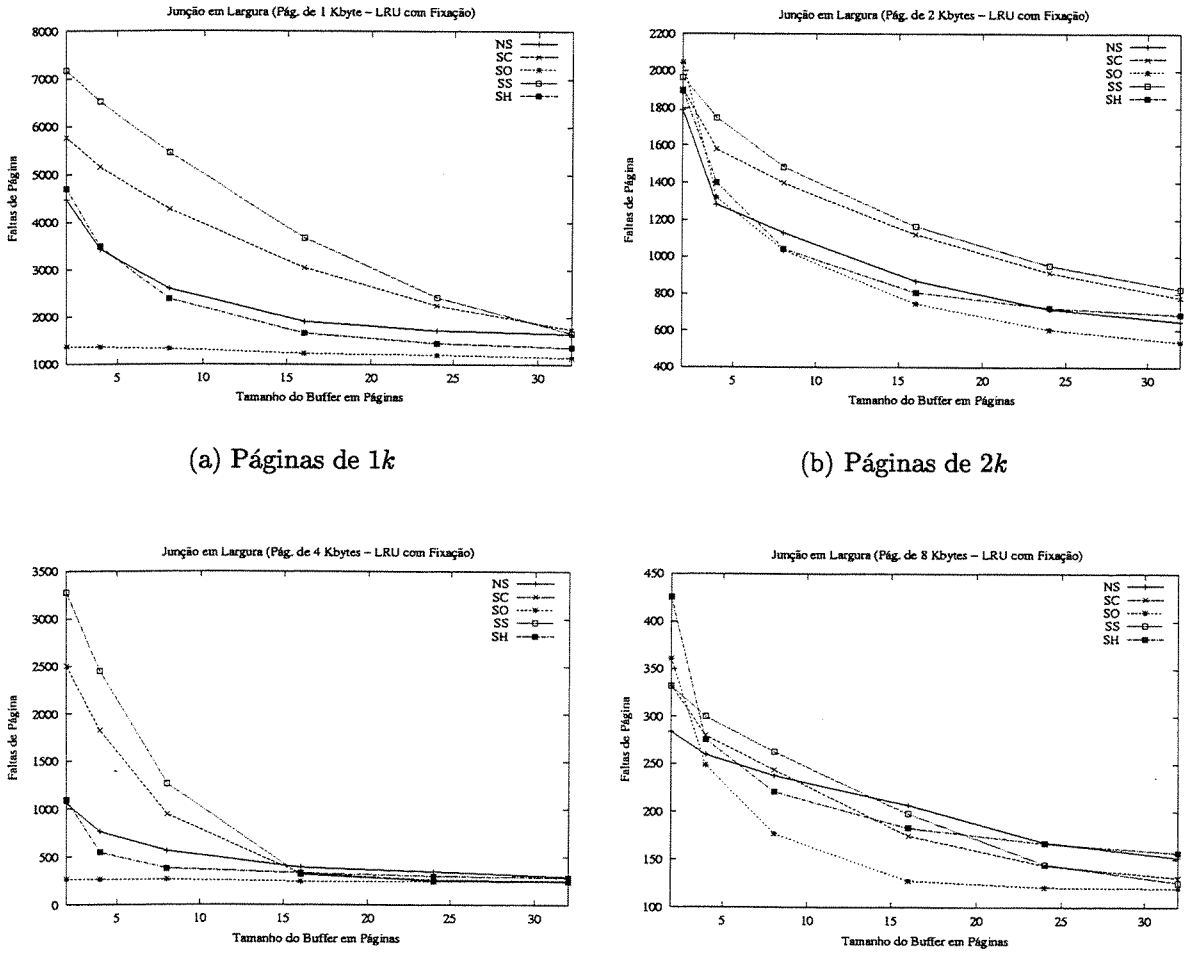



(c) Páginas de $4k$




(d) Páginas de $8k$

Figura 4.13: Resultados obtidos com o método BF, política LRU sem fixação, executando sobre índices de $1k$ (a), $2k$ (b), $4k$ (c) e $8k$ (d).




(a) Páginas de $1k$

(b) Páginas de $2k$

(c) Páginas de $4k$

(d) Páginas de $8k$

Figura 4.14: Resultados obtidos com o método BF, política LRU com fixação, executando sobre índices de $1k$ (a), $2k$ (b), $4k$ (c) e $8k$ (d).

número de acessos a disco, é tremendamente superior em relação a seus concorrentes. Isto se deve justamente à capacidade de se saber de antemão quais páginas são necessárias a um bom arranjo na seqüência de leituras das mesmas.

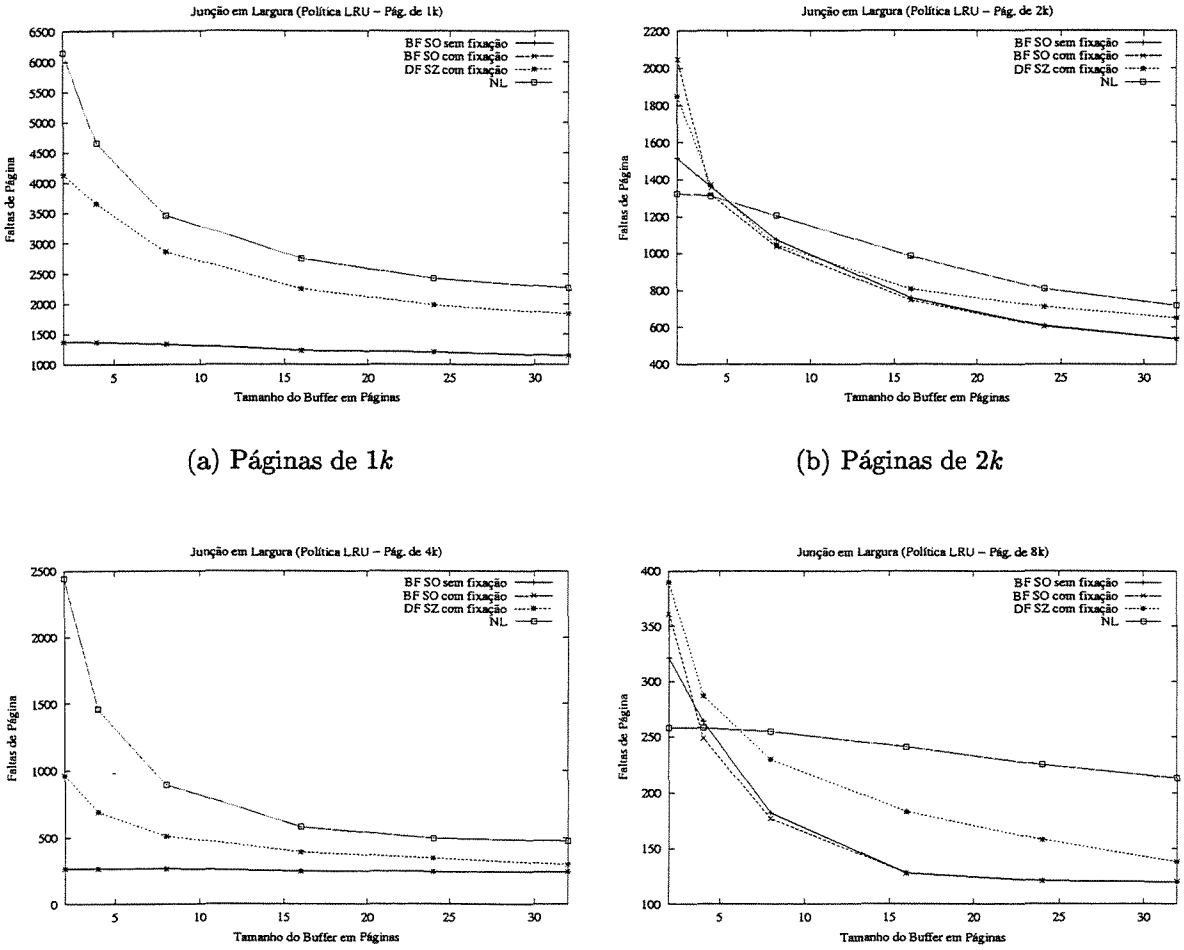
## 4.7 Outros resultados

Além dos experimentos mostrados aqui, onde efetuamos a junção sobre o conjunto de postes e quadras, realizamos outros testes. Estes testes foram as junções *quadras* × *quadras* (QQ), *quadras* × *cidade* (QC), *cidade* × *poste* (CP) e *cidade* × *cidade* (CC). Os resultados da execução dos testes QQ, QC e CP refletem os resultados da junção de quadras e postes. Entretanto, os resultados dos testes CC divergiram dos primeiros.

Segundo os testes CC, o método de junção NL foi superior ao método DF, em relação ao número de operações de I/O, embora o tempo de CPU gasto fosse bem acima do obtido com os métodos DF e BF sem ordenação. O método BF continuou tendo desempenho melhor, mas quando o tamanho do *bufferpool* era pequeno em relação à quantidade de páginas existentes em disco, o método NL se tornava melhor.

O conjunto de dados *cidade* caracteriza-se por possuir grande número de elementos, de alta densidade e objetos com tamanhos muito diferenciados, variando desde pontos até MBRs que cobrem boa parte da área de dados. Estes MBRs, no processo de inserção, geram MBRs de níveis internos grandes, que se propagam até a raiz. Este fato gera um grande número de interseções entre os MBR dos níveis internos, o que pode ser visto na Figura 4.16, que mostra os MBRs referentes ao nível 3, o último nível de nós não-folha, para o índice de 2 *kbytes*.

Neste contexto, os resultados referentes ao conjunto *cidades* podem ser causados pela variância com relação aos dados. Isto também está relacionado com a rotina de inserção dos dados no índice: se a rotina de inserção gera níveis internos com muita sobreposição, degradará o desempenho das consultas em geral. Entretanto, para se afirmar sem dúvida a causa desta anomalia, seria necessário realizar outros experimentos, com dados de configurações variadas.




(a) Páginas de $1k$

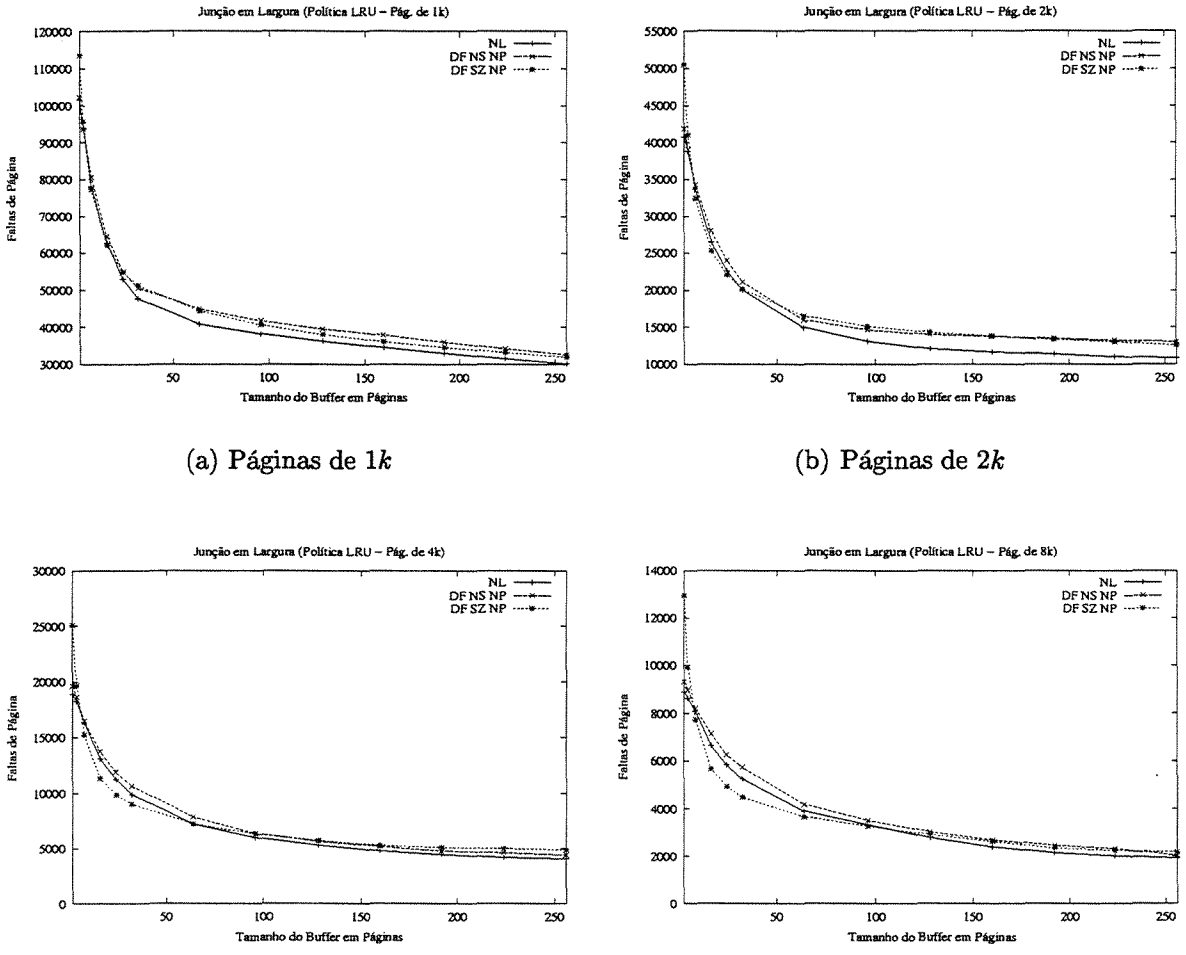
(b) Páginas de $2k$

(c) Páginas de $4k$

(d) Páginas de $8k$

Figura 4.15: Comparação entre o método BF SO com e sem fixação, método DF SZ sem fixação e o método NL, em páginas de $1k$ (a), $2k$ (b), $4k$ (c) e $8k$ (d).

Figura 4.16: MBRs internos correspondentes ao nível 3 do índice gerado com páginas de *2 kbytes*, para o conjunto *cidade*.




(a) Páginas de $1k$

(b) Páginas de $2k$

(c) Páginas de $4k$

(d) Páginas de $8k$

Figura 4.17: Desempenho do método NL e DF NS e SZ sem fixação, utilizando *cidade* em páginas de $1k$ (a), $2k$ (b), $4k$ (c) e $8k$ (d).

# Capítulo 5

# Conclusões e extensões

Este trabalho contemplou a análise de desempenho de junções espaciais com relação ao tamanho da página de disco, tamanho de *buffer pool*, critérios de ordenação de índices intermediários de junção, políticas de substituição de páginas, utilização de memória, tempo de CPU e número de operações de E/S.

Para este fim, implementamos um sistema modularizado que oferece total controle sobre a operação de junção. Este sistema incorpora três operações de junção espacial, baseadas em percurso sincronizado em árvores, a saber: *nested-loop* (NL), *depth-first* (DF) e *breadth-first* (BF). O índice adotado é a R*-tree mas, com poucas alterações, é possível incluir qualquer método de acesso espacial.

Os testes de junção foram executados sobre dados reais extraídos de uma aplicação de telefonia, também representativos em aplicações de utilidade pública como distribuição de energia elétrica e água e esgoto, dentre outras.

Através dos resultados dos testes, podemos concluir que:

- o aumento da página se traduz em aumento da carga de CPU. Desta forma, o aumento da página deve ser acompanhado pelo aperfeiçoamento dos algoritmos de *plane-sweeping*;

- uma boa ordenação dos IIJs é fundamental ao bom desempenho da junção. Em nossos experimentos, as melhores foram a SZ sem fixação de páginas (método DF) e a SO com fixação (método BF);

- o bom desempenho das junções espaciais também depende da existência de *buffer-pools* com uma boa proporção de páginas. Em nossos experimentos, uma boa configuração foi um *bufferpool* com aproximadamente 10% ou mais das páginas existentes no disco;

- a utilização do algoritmo BF pode, dependendo da densidade dos dados, resultar em índices intermediários de tamanhos proibitivos;
- há conjuntos de dados em que a utilização do método DF resulta em desempenho pior, quanto ao número de operações de entrada e saída, que o método NL;
- as rotinas de inserção e manipulação da árvore têm fundamental importância no desempenho das consultas em geral.

Além da parte experimental, este trabalho executou pesquisas sobre outros métodos espaciais. Outros estudos efetuados foram sobre modelos analíticos de desempenho de junções espaciais.

É possível perceber que os métodos mais considerados nos trabalhos sobre indexação e consultas espaciais são os baseados em R-trees. Estes métodos são relativamente simples de serem implementados, e possuem as características inerentes a métodos baseados em árvores multiárias, que as tornam adequadas ao armazenamento secundário.

Entretanto, sentimos a falta de estudos sobre a influência da variância dos dados no desempenho dos métodos, bem como a validação dos resultados experimentais em relação aos modelos analíticos desenvolvidos, tanto em relação à junção espacial quanto a outras consultas e à indexação.

Podemos destacar como extensões a este trabalho:

- comparação com os resultados dos testes com resultados de métodos analíticos de desempenho;
- a incorporação e avaliação de novos algoritmos de junção, como os métodos desenvolvidos para realização de junção onde há entradas não indexadas;
- aperfeiçoamento dos algoritmos quanto ao aspecto CPU;
- novas investigações sobre o desempenho dos métodos, em especial o método DF, com variadas configurações de dados;
- investigação de métodos que ataquem a fase de refinamento;
- investigação da junção em que consideramos estruturas de indexação distintas;
- investigação do desempenho da junção espacial utilizando outros métodos de indexação como base;
- investigação do desempenho da junção espacial utilizando outros predicados de junção.

Métodos de acesso espaciais, mesmo após vários anos de pesquisa, ainda possuem setores que necessitam de maior amadurecimento. Isto pode ser observado principalmente quando falamos de junções espaciais.

Em resumo, métodos de junção espacial podem e devem ser aperfeiçoados, levando-se em conta os fatores aqui analisados. Os ganhos que os ajustes destes fatores podem trazer justificam plenamente este trabalho e novas pesquisas.

# Bibliografia


[AGPZ99] D. J. Abel, V. Gaede, R. Power, and X. Zhou. Caching strategies for spatial joins. *GeoInformatica*, 3(1), March 1999.

[AHVV99] L. Arge, K. H. Hinrichs, J. Vahrenhold, and J. S. Vitter. Efficient bulk operations on dynamic *R*-trees. *Lecture Notes in Computer Science*, 1619:328–348, 1999.

[APR+98] L. Arge, O. Procopiuc, S. Ramaswamy, T. Suel, and J. Vitter. Scalable sweeping-based spatial join. In *Proc. of the 24th VLDB Conference*, New York, USA, 1998.

[Arg95] L. Arge. The buffer tree: A new technique for optimal I/O-algorithms. *Lecture Notes in Computer Science*, 955, 1995.

[AT97] C. H. Ang and T. C. Tan. New linear node splitting algorithm for R-trees. *Lecture Notes in Computer Science*, 1262, 1997.

[Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[BK94] T. Brinkhoff and H. Kriegel. Aproximations for a multi-step processing of spatial joins. *Lecture Notes in Computer Science*, 1994.

[BKK96] S. Berchtold, D. A. Keim, and H. P. Kriegel. The X-tree : An index structure for high-dimensional data. In *VLDB'96, Proceedings of 22th International Conference on Very Large Data Bases*, pages 28–39, Bombaim, India, September 1996. Morgan Kaufmann.

[BKS93] T. Brinkhoff, H.-P. Kriegel, and B. Seeger. Efficient processing of spatial joins using R-trees. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 22(2):237–246, June 1993.

[BKSS90] N. Beckmann, H. Kriegel, R. Schneider, and B. Seeger. The R*-tree: an efficient and robust access method for points and rectangles. In *Proceeding of the 1990 ACM SIGMOD International Conference on Management of Data*, June 1990.

[BKSS94] T. Brinkhoff, H. Kriegel, R. Schneider, and B. Seeger. Multi-step processsing of spatial joins. In *Proceedings of the 1994 ACM SIGMOD*, SIGMOD RECORD. ACM Press, May 1994.

[BSW97] J. Bercken, B. Seeger, and P. Widmayer. A generic approach to bulk loading multidimensional index structures. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pages 406–415, 1997.

[Car98] A. P. Carneiro. Análise de desempenho de métodos de acesso espaciais baseada em um banco de dados real. Master's thesis, Universidade Estadual de Campinas - UNICAMP, 1998.

[CCH+96] G. Câmara, M. A. Casanova, A. S. Hemerly, G. C. Magalhães, and C. M. B. Medeiros. *Anatomia de Sistemas de Informação Geográfica.* 1996.

[CCR98] L. Chen, R. Choubey, and E. Rundensteiner. STLT: Bulk insertion into R-trees. In *A poster paper in the 6th ACM Workshop on Geographic Information Systems (ACM-GIS'98)*, Washington, DC, November 1998. http://davis.wpi.edu/dsrg/dsrg-all-pubs.html.

[CM92] F. S. Cox and G. C. Magalhães. Implementação e análise de métodos de acesso adados espaciais. In *VII Simpósio Brasileiro de Banco de Dados*, Porto Alegre, Maio 1992.

[dS99] G. Z. da Silva. *Avaliação de Junções em Bancos de Dados Espaciais.* PhD thesis, COPPE/UFRJ, Rio de Janeiro, Junho 1999.

[Fal86] C. Faloutsos. Multiattribute hashing using Gray codes. In *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 227–238, Washington, D.C., May 1986.

[FB74] R. A. Finkel and J. L. Bentley. Quad trees: A data structure for retrieval on composite keys. *Acta Informatica*, 4:1–9, 1974.

[FR89] C. Faloutsos and S. Roseman. Fractals for secondary key retrieval. In *Proceedings of the Eighth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems (PODS '89)*, pages 247–252, Philadelphia, PA, USA, March 1989. ACM Press.

[FR91] C. Faloutsos and Y. Rong. DOT: A spatial access method using fractals. In *International Conference on Data Engineering*, pages 152–159, Los Alamitos, Ca., USA, April 1991. IEEE Computer Society Press.

[GLL97] Y. J. García, M. A. López, and S. T. Leutenegger. A greedy algorithm for bulk loading R-trees. Technical Report TR-97-02, Department of Math and Computer Science, University of Denver, 1997. Disponível no URL http://www.cs.du.edu/ leut/pubs.html.

[GLL98] Y. García, M. López, and S. Leutenegger. On optimal node splitting for R-tree. In *Proceedings of the 24th VLDB Conference*, 1998. Disponível no URL http://www.cs.du.edu/ leut/pubs.html.

[GOP+98] O. Günther, V. Oria, P. Picouet, J.-M. Saglio, and M. Scholl. Benchmarking spatial joins à la carte. In *Proceedings of IEEE SSDBM*, Capri – Italy, July 1998.

[Gre89] D. Greene. An implementation and performance analysis of spatial data access methods. In *Proceeding of the Fifth International Conference on Data Engineering*, pages 606–615, February 1989.

[Gün93] O. Günther. Efficient computation of spatial joins. In *International Conference on Data Engineering*, pages 50–60, Los Alamitos, Ca., USA, Abril 1993. IEEE Computer Society Press.

[Gut84] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM SIGMOD Conference on Management of Data*, 1984.

[HJR97] Y.-W. Huang, N. J., and E. A. Rundensteiner. Spatial joins using R-trees: Breadth-first traversal with global optimizations. In *VLDB'97, Proceedings of 23rd International Conference on Very Large Data Bases*, pages 396–405, 1997.

[HS95] E. G. Hoel and H. Samet. Benchmarking spatial joins operations with spatial output. In *Proc. of the 21th Intl. Conf. on Very Large Databases*, Zurich – Switzerland, November 1995.

[Jag90] H. V. Jagadish. Linear clustering of objects with multiple attributes. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 19(2):332–342, June 1990.

[KF92] I. Kamel and C. Faloutsos. Parallel R-trees. In M. Stonebraker, editor, *Proceedings of the 1992 ACM SIGMOD International Conference on Management of Data*, San Diego, California, June 1992. ACM Press.

[KF93] I. Kamel and C. Faloutsos. On packing R-trees. In *Proceedings of the 2nd International Conference on Information and Knowledge Management - CKIM-93*, November 1993.

[KF94] I. Kamel and C. Faloutsos. Hilbert R-tree: an improved R-tree using fractals. In *Proceedings of the 20th VLDB Conference*, pages 500–509, September 1994.

[KS97a] N. Katayama and S. Satoh. The SR-tree: An index structure for high-dimensional nearest neighbor queries. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2), 1997.

[KS97b] N. Koudas and K. C. Sevcik. Size separation spatial join. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, SIGMOD Record, pages 324–335, New York, May 1997.

[KSSS89] H. P. Kriegel, M. Schiwietz, R. Schneider, and B. Seeger. Performance comparison of point and spatial access methods. In *Proceedings of the 1st Symposium on Design and Implementation of Large Spatial Databases*, volume 409 of *Lecture Notes in Computer Science*, Berlin, 1989. Springer-Verlag.

[Kum94] A. Kumar. G-tree: A new data structure for organizing multidimensional data. *Transactions on Knowledge and Data Engineering*, 6(2):341–347, April 1994.

[LLE97] S. Leutenegger, M. López, and J. Edginton. STR: a simple and efficient algorithm for R-tree packing. In *Proceeding of the 13th International Conference on Data Engineering*, New York, USA, April 1997.

[LLRC96] Y.-J. Lee, D.-M. Lee, S.-J. Ryu, and C.-W. Chung. Controlled decomposition strategy for complex spatial objects. *Lecture Notes in Computer Science*, 1134:207–223, 1996.

[LR94] M. Lo and C. V. Ravishankar. Spatial joins using seeded trees. In *Proceedings of the 1994 ACM SIGMOD*, volume 23 of *SIGMOD RECORD*, pages 209–220. ACM Press, June 1994.

[LR96] M. Lo and C. V. Ravishankar. Spatial hash-joins. In *Proceedings of the 1996 ACM SIGMOD*, SIGMOD RECORD, pages 247–258, Montreal, Canada, June 1996. ACM Press.

[Mag93] G. C. Magalhães. Projeto SAGRE. *Fator GIS*, Out./Nov./Dez. 1993.

[Mag94] G. C. Magalhães. The development of open systems for engineering applications. In *Proc. of XVII Intl. Conference on AM/FM*, Denver - Colorado, March 1994.

[Mag97] G. C. Magalhães. Telecommunications outside plant managements throughout Brazil. In *Conference XX Proceendings*, Nashville, 1997.

[MCD94] M. R. Mediano, M. A. Casanova, and M. Dreux. V-trees — A storage method for long vector data. In *20th International Conference on Very Large Databases*, September 1994.

[MGS+94] G. C. Magalhães, A. V. Gigliotti, C. L. F. Santos, D. Teijeiro, and E. L. Argondizio. Especificação técnica da conversão de dados - Proposta da Telebrás - Projeto SAGRE. In *GIS Brasil 94*, Curitiba - Paraná, Outubro 1994.

[MP99] N. Mamoulis and D. Papadias. Integration of spatial join algorithms for processing multiple inputs. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, Filadelphia – Pensilvania, June 1999.

[ND97] M. Nascimento and M. Dunham. Using B+-trees in a two disk-single processor architecture to efficiently process inclusion spatial queries. In *Proceedings of the 5th International Workshop on Advances in Geographic Information Systems (GIS-97)*, pages 5–8, New York, November 1997. ACM Press.

[NDK96] M. Nascimento, M. Dunham, and V. Kouramajian. A multiple tree mapping-based approach for range indexing. *Journal of the Brazilian Computer Society*, April 1996.

[NHS84] J. Nievergelt, H. Hinterberger, and K. C. Sevcik. The grid file: An adaptable, symmetric multikey file structure. *ACM Transactions on Database Systems*, 9(1):38–71, March 1984.

[OM84] J. A. Orenstein and T. H. Merrett. A class of data structures for associative searching. In *Proceedings of the Fourth ACM SIGACT-SIGMOD Symposium on Principles of Database Systems*, pages 181–190, Portland, Oregon, March 1984.

[Ooi90] B. C. Ooi. *Efficient query processing in geographic information systems*, volume 471 of *Lecture Notes in Computer Science*. Springer-Verlag Inc., New York, NY, USA, 1990.

[OSH93] B. C. Ooi, R. Sacks-Davis, and J. Han. Indexing in spatial databases. 1993. Non-published paper, available at http://www.comp.nus.edu.sg/ ooibc/papers/survey.ps.

[PD96] J. Patel and D. DeWitt. Partition based spatial-merge join. In *SIGMOD*, pages 259 – 270, Montreal, Canada, June 1996.

[PMT99] D. Papadias, N. Mamoulis, and Y. Theodoridis. Processing and optimization of multiway spatial joins using R-tree. In *Proc. of the ACM SIGACT-SIGMOD-SIGART Symposium of Database Systems*, 1999.

[PRS99] A. Papadopoulos, P. Rigaux, and M. Scholl. A performance evaluation of spatial join processing strategies. *Lecture Notes in Computer Science*, 1651, 1999.

[RL85] N. Roussopoulos and D. Leikfer. Direct spatial search on pictorial databases using packed R-tree. In *Proceeding of ACM SIGMOD*, May 1985.

[SK91] R. Schneider and H. P. Kriegel. The TR*–tree: A new representation of polygonal objects supporting spatial queries and operations. In *Proceedings of Computational Geometry (CG '91)*, volume 553 of *LNCS*, pages 249–264, Berlin, Germany, March 1991. Springer.

[SK96] K. C. Sevcik and N. Koudas. Filter trees for managing data over a range of size granularities. Technical report, Computer Systems Research Institute – University of Toronto, 1996.

[SL99] B. Schnitzer and S. T. Leutenegger. Master-Client R-trees: A new parallel R-tree architecture. In *Proceedings of the 11th International Conference on Scientific and Statistical Database Management (SSDBM 99)*, 1999. A publicar, disponível no URL http://www.cs.du.edu/ leut/pubs.html.

[SLS95] E. Stefanakis, Y. C. Lee, and T. Sellis. The abstraction technique for spatial access method. Technical Report KDBSLAB-TR-95-01, National Technical University of Athens, February 1995. http://www.dbnet.ece.ntua.gr/publications/technical_reports.html.

[SRF87] T. Sellis, N. Roussopoulos, and C. Faloutsos. The $R^+$-tree: a dynamic index for multi-dimensional objects. In *Proceedings of the 13th VLDB Conference*, pages 507 – 518, Brighton – England, September 1987.

[TS93] Y. Theodoridis and T. Sellis. Optimization issues in R-tree construction. Technical Report KDBSLAB-TR-93-08, National Technical University of Athens, Athens – Greece, October 1993. http://www.dbnet.ece.ntua.gr/publications/technical_reports.html.

[TS95] Y. Theodoridis and T. Sellis. A model for the prediction of R-tree performance. Technical Report KDBSLAB-TR-95-10, National Technical University of Athens, Athens – Greece, Outubro 1995. http://www.dbnet.ece.ntua.gr/publications/technical_reports.html.

[TSS97] Y. Theodoridis, E. Stefanakis, and T. Sellis. An efficient cost model for spatial joins using R-trees. Technical Report KDBSLAB-TR-97-01, National Technical University of Athens, Athens – Greece, February 1997. Disponível no URL http://www.dbnet.ece.ntua.gr/publications/technical_reports.html.

[TSS98] Y. Theodoridis, E. Stefanakis, and T. Sellis. Cost models for join queries in spatial databases. In *Proc. of the IEEE International Conference on Data Engineering (ICDE)*, Orlando – Florida, February 1998.

[TSS99] Y. Theodoridis, E. Stefanakis, and T. Sellis. Efficient cost models for spatial queries using R-trees. *IEEE Transactions on Knowledge and Data Engineering*, 1999. To appear.

[Val87] P. Valduriez. Join indices. *ACM Transactions on Database Systems*, 12(2), 1987.

[WJ96] D. A. White and R. Jain. Similarity indexing with the SS-tree. In *Proceedings of the 12th International Conference on Data Engineering*, pages 516–523. IEEE Computer Society, February 1996.

[ZdS98] G. Zimbrão and J. M. de Souza. A raster aproximation for the processing of spatial joins. In *Proc. os the 24th VLDB Conference*, New York, USA, August 1998.