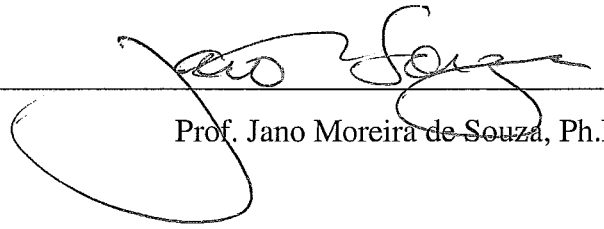
REALIZAÇÃO EFICIENTE DE CONSULTAS EM BANCOS DE DADOS ESPAÇOTEMPORAIS

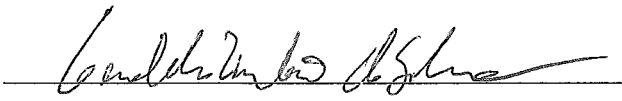
Victor Teixeira de Almeida

TESE SUBMETIDA AO CORPO DOCENTE DA COORDENAÇÃO DOS PROGRAMAS DE PÓS-GRADUAÇÃO DE ENGENHARIA DA UNIVERSIDADE FEDERAL DO RIO DE JANEIRO COMO PARTE DOS REQUISITOS NECESSÁRIOS PARA A OBTENÇÃO DO GRAU DE MESTRE EM CIÊNCIAS EM ENGENHARIA DE SISTEMAS E COMPUTAÇÃO.

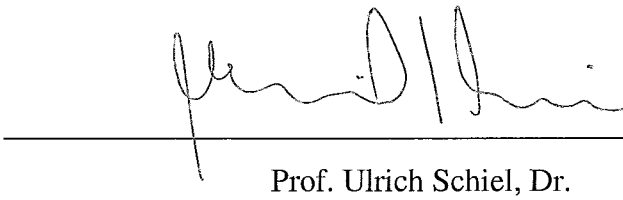
Aprovada por:



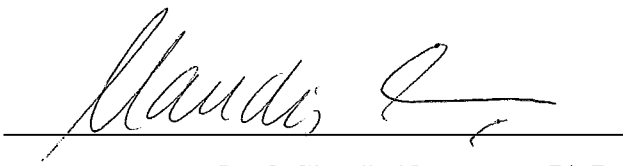
Prof. Jano Moreira de Souza, Ph.D.



Prof. Geraldo Zimbrão da Silva, D.Sc.



Prof. Ulrich Schiel, Dr.



Prof. Claudio Esperança, Ph.D.

RIO DE JANEIRO, RJ - BRASIL

SETEMBRO DE 2001

ALMEIDA, VICTOR TEIXEIRA DE

Realização Eficiente de Consultas em Bancos de Dados Espaçotemporais [Rio de Janeiro] 2001.

IX, 88 p. 29,7 cm (COPPE/UFRJ, M.Sc., Engenharia de Sistemas e Computação, 2001)

Tese - Universidade Federal do Rio de Janeiro, COPPE

1. Banco de Dados Espaçotemporais

I. COPPE/UFRJ II. Título (série)

Agradecimentos

Aos meus pais e irmãos, que sempre me conduziram para os caminhos corretos na vida.

À Gisele Reis, pelo incansável apoio e dedicação indispensáveis à realização deste trabalho.

Ao Prof. Jano de Souza, por suas sábias orientações que decidiram o rumo de minha vida acadêmica e profissional.

Ao Prof. Geraldo Zimbrão, por ser mais do que um orientador, um amigo. Obrigado também pela ajuda na fase de implementação.

À Patrícia Leal, por estar sempre presente e disposta a ajudar, não deixando que nada me faltasse durante a execução deste trabalho.

Ao Prof. Cláudio Esperança, pelo código da R*-Tree. Sem ele o trabalho de implementação seria mais árduo.

À Profa. Marta Mattoso pelo carinho e dedicação nos ensinamentos desde minha graduação.

À Renata Wo, por seus auxílios na etapa de implementação, principalmente nos algoritmos da R*-Tree.

À CAPES, pela concessão da bolsa de estudos para que eu pudesse me dedicar à este trabalho.

A todas as outras pessoas que, direta ou indiretamente, contribuíram para a execução deste trabalho.

Resumo da Tese apresentada à COPPE/UFRJ como parte dos requisitos necessários para a obtenção do grau de Mestre em Ciências (M.Sc.)

REALIZAÇÃO EFICIENTE DE CONSULTAS EM BANCOS DE DADOS ESPAÇOTEMPORAIS

Victor Teixeira de Almeida

Setembro/2001

Orientadores: Jano Moreira de Souza
Geraldo Zimbrão da Silva

Programa: Engenharia de Sistemas e Computação

É um fato conhecido que as novas aplicações de Sistemas de Informações Geográficas (SIG) necessitam armazenar informações temporais. Entretanto, os índices espaciais mais utilizados, a R-Tree e suas variantes, não preservam a evolução dos objetos espaciais. Alguns métodos de acesso foram propostos na literatura e permitem a recuperação de estados presentes e passados dos dados. Nesta dissertação avaliaremos o desempenho dos métodos de acesso mais populares para executar consultas espaçotemporais. Por consultas entende-se consultas de seleção bem como consultas de junção. Consultas espaçotemporais podem ser agrupadas em duas classes: consultas envolvendo instantes de tempo e consultas envolvendo intervalos de tempo. Propomos uma nova estrutura de indexação de dados espaçotemporais chamada TR-Tree (do inglês Temporal R-Tree) que combina características temporais da Multiversion B-Tree com as características espaciais da R*-Tree. Projetamos e implementamos os algoritmos de inserção, remoção, consultas de seleção e consultas de junção. Esta dissertação é, até onde sabemos, um estudo pioneiro em junções espaçotemporais. Foi feito um estudo comparativo com diversos conjuntos de dados para mostrar a escalabilidade dos algoritmos da TR-Tree. Nossos experimentos mostraram que seu desempenho e utilização de espaço são muito bons até mesmo nos casos extremos, mostrando que a TR-Tree é atualmente a melhor escolha para a indexação de dados espaçotemporais.

Abstract of Thesis presented to COPPE/UFRJ as a partial fulfillment of the requirements for the degree of Master of Science (M.Sc.)

EFFICIENT QUERY PROCESSING FOR SPATIOTEMPORAL DATABASES

Victor Teixeira de Almeida

September/2001

Advisors: Jano Moreira de Souza
Geraldo Zimbrão da Silva

Department: Systems and Computing Engineering

It's a well-known fact that new GIS applications need to keep track of temporal information. However, the most widely used spatial indexes, the R-Tree and its variants, do not preserve the evolution of bounding boxes. Some new indexing structures were proposed in the literature that allow the retrieval of present and past states of data. In this thesis we evaluate the most popular index structures to perform spatiotemporal queries. As queries we mean selection queries and join queries. Spatiotemporal queries can be grouped in two classes: queries involving time instants and queries involving time intervals. We propose a new index structure termed Temporal R-Tree, which combines several features of Becker's Multi-Version B-Tree with the spatial indexing characteristics of the R*-Tree. We have designed and implemented algorithms for insertion, deletion, selection queries, and join queries. This thesis is, at the best of our knowledge, the first study on spatiotemporal join queries. A number of comparative queries on several data sets were performed to show the scalability of the Temporal R-Tree. Our experiments show that its performance and space utilization are very good even on worst cases, showing that the Temporal R-Tree is by now the best choice for indexing spatiotemporal data.

Índice

Índice de Figuras

Índice de Tabelas

1. Introdução

Em 1905 Einstein publicava a Teoria da Relatividade onde, dentre outras noções, soldava o espaço e o tempo juntos em uma única abstrata noção quadri-dimensional chamada espaço-tempo. Para ele, em sua teoria, as noções de espaço e tempo não são absolutas, ou seja, a taxa de tiques de um relógio depende da velocidade do observador do relógio. Isto contrariava a idéia de que o tempo era absoluto sendo que as noções de espaço e tempo (onde e quando) não puderam mais ser tratadas de forma separada.

A noção que Einstein tinha sobre o espaço e o tempo e que ele chamava espaço-tempo, é o que estamos tentando representar, armazenar e recuperar em Bancos de Dados, e o modificador "espaçotemporal" vem à tona para identificar que estamos lidando com estes tipos de dados. Nesse contexto surgem os Bancos de Dados Espaçotemporais (BDET), que são bancos de dados capazes de lidar com informações espaçotemporais.

Nesta dissertação serão abordados assuntos sobre a área de Bancos de Dados Espaçotemporais (BDET) e nossa análise será concentrada no armazenamento e recuperação eficientes de dados espaçotemporais, considerando principalmente a indexação destes dados e seu impacto sobre as principais consultas de BDETs.

1.1. Motivação

As pesquisas das áreas de Bancos de Dados Espaciais (BDE) e Sistemas de Informações Geográficas (SIG) estão refletindo uma necessidade emergente de se modelar objetos geográficos ou espaciais ao longo do tempo. O mundo é dinâmico e esta realidade tem que ser refletida pelos sistemas que o modelam. Os objetos espaciais crescem, movem-se e mudam de formato ao longo do tempo e estas modificações dos objetos têm que ser armazenadas. De outra forma, informações relevantes como formato, velocidade, direção e sentido podem ser escondidas da visão dos usuários destes sistemas.

Como exemplo de aplicações do mundo real que necessitam do armazenamento temporal dos objetos espaciais podemos citar sistemas de monitoração de incêndios, furacões, ciclones ou quaisquer outras intempéries da natureza; sistemas simuladores de vôo; sistemas de previsão de tempo que utilizam-se de fotos de satélite; e muitos outros. Todas essas aplicações têm em comum o movimento de objetos ao longo do tempo e o fato de que se forem utilizadas técnicas convencionais de armazenamento dos dados espaciais, toda a informação temporal dos objetos será perdida.

Neste sentido, surgem novos ramos de pesquisa como Bancos de Dados Temporais (BDT) (SNODGRASS, 1992), Bancos de Dados Espaçotemporais (BDET) (ABRAHAM et al., 1999) e Sistemas de Informações Geográficas Temporais (SIGT) (LANGRAN, 1992). Estas novas aplicações tendem a manter a informação temporal dos objetos e diferem entre si quando se tratam de objetos espaciais (BDET e SIGT) ou escalares (BDT). A evolução dos objetos espaciais ou escalares ao longo do tempo, neste tipo de aplicação, se mantém armazenada e consultas podem ser feitas no presente, passado e futuro (previsão). Ou seja, a base de dados não reflete mais somente uma foto dos objetos no momento atual, mas um completo e vasto histórico de objetos contendo suas modificações ao longo do tempo.

As aplicações puramente espaciais (BDE e SIG) por si só já possuem uma alta complexidade quanto ao armazenamento e recuperação dos dados através de consultas espaciais. Uma vez adicionada mais uma dimensão, o tempo, estas aplicações se tornam ainda mais complexas pois as consultas agora envolvem também esta nova dimensão e o volume de dados se torna ainda maior, dado que não mais se remove objetos da base de dados. Surge então a necessidade de novos métodos para armazenamento e recuperação eficientes desses tipos de dados e é nesse contexto que se insere essa dissertação.

1.2. Definindo o Problema

Bancos de Dados Espaciais permitem o armazenamento de objetos espaciais como pontos, linhas, polígonos, etc. e ainda a pesquisa e recuperação eficiente destes objetos a partir de suas posições e formatos espaciais. Estruturas eficientes de indexação através da posição e formato espacial dos objetos constituem um campo crescente de pesquisa. Os principais SGBDs como ORACLE, Informix, DB2, etc. já provêm

métodos de acesso a dados espaciais eficientes através da utilização de índices espaciais. Em Bancos de Dados Convencionais, em geral, a utilização de índices sobre as relações acelera o processo de recuperação de dados. Em BDE, devido a uma maior complexidade das consultas, a utilização de índices não serve só para acelerar as consultas, mas para torná-las viáveis, uma vez que muitas consultas não são viáveis de serem executadas com tempo de resposta razoável sem a utilização de um índice espacial.

Com a necessidade de informações temporais sobre objetos espaciais, estes índices precisam ser adaptados para suportar consultas espaçotemporais, ou seja, que envolvam posição espacial e tempo como parâmetros de restrição utilizados conjuntamente. Como objetos nestas bases de dados nunca são removidos, a quantidade de informação armazenada por este tipo de aplicação torna-se ainda maior e um índice, neste caso, torna-se ainda mais necessário. Apenas protótipos de SGBDs Espaçotemporais, como SGBDs Temporais, podem ser encontrados atualmente.

Por consultas espaciais entende-se as consultas que envolvem a posição e formato espacial dos objetos como restrição da consulta, como por exemplo a interseção, proximidade, pertinência, etc. Por consultas temporais entende-se as consultas que envolvem o tempo de validade dos objetos como restrição da consulta, como por exemplo consultas por um determinado instante de tempo ou por um intervalo de tempo. Desta forma, consultas espaçotemporais são aquelas que envolvem ambos os tipos de restrição conjuntamente: espacial e temporal.

A dificuldade de implementação de métodos de acesso espaçotemporais encontra-se na integração das dimensões espaço e tempo, que separadamente já constituem complexos campos de pesquisa com muito trabalho em desenvolvimento.

1.3. Contribuições desta Tese

Nesse contexto de Bancos de Dados Espaçotemporais iremos, nesta dissertação, propor e implementar um novo método de acesso eficiente a dados espaçotemporais, a TR-Tree. A TR-Tree suporta os principais tipos de consultas espaçotemporais: as consultas de seleção e a junção espaçotemporal. Serão propostos os principais algoritmos de inserção, remoção, consultas de seleção e junção espaçotemporal para a

TR-Tree, bem como uma avaliação experimental comparando-a a alguns métodos de acesso existentes na literatura através de diversos conjuntos de dados e consultas espaçotemporais.

1.4. Estrutura da Tese

Esta dissertação encontra-se organizada em quatro capítulos além desta introdução. O capítulo 2 apresenta conceitos relevantes à área de estudo desta dissertação bem como um estudo técnico da necessidade da utilização de métodos de acesso para dados espaçotemporais, definindo os principais tipos de consultas espaçotemporais. O Capítulo 3 apresenta uma revisão bibliográfica dos métodos de acesso existentes na literatura relevantes. O capítulo 4 apresenta a proposta do novo método de acesso espaçotemporal com seus algoritmos principais de inserção, remoção, consultas de seleção e junção espaçotemporal. No capítulo 5 faremos uma comparação experimental entre alguns métodos de acesso espaçotemporais citados no Capítulo 3 que foram implementados para a realização deste trabalho. O capítulo 6 fecha esta dissertação com nossas conclusões finais e propostas de trabalhos futuros.

2. Consultas Espaçotemporais

Este capítulo destina-se a apresentar os principais tipos de consultas que um Banco de Dados Espaçotemporal deve suportar. Alguns conceitos iniciais sobre Banco de Dados Temporais (BDT) serão apresentados na Seção 2.1 e, em seguida, um detalhamento das principais consultas utilizadas em Bancos de Dados Espaciais (BDE) será feito na Seção 2.2. Uma extensão destas consultas levando-se em consideração o tempo será mostrada na Seção 2.3 resultando nas principais consultas de Bancos de Dados Espaçotemporais (BDET).

2.1. Principais Conceitos de Bancos de Dados Temporais

Em 1998, a comunidade de BDTs se reuniu para tentar padronizar a nomenclatura para os conceitos mais amplamente encontrados na literatura (JENSEN et al., 1998)[1]. Vários conceitos de uso comum eram citados em diferentes trabalhos com diferentes nomes. Neste trabalho encontram-se estes conceitos definidos explicitamente e com apenas um nome para cada um decidido em comum acordo pelos membros da comunidade de BDTs.

Dentre os conceitos descritos em JENSEN et al. (1998) encontramos alguns de fundamental importância para esta dissertação e iremos detalhá-los a seguir. As definições diretamente retiradas do artigo original encontram-se entre aspas e em itálico, seguindo abaixo um complemento à definição original.

2.1.1. Tempo de Validade

"*O tempo de validade de um fato é o tempo em que o fato é verdadeiro na realidade modelada*".

O tempo de validade de um fato no banco de dados é uma dimensão temporal criada pelo usuário para modelar a temporalidade deste fato. Um exemplo pode esclarecer melhor este conceito: o tempo de validade de um cartão de crédito normalmente é de cinco anos inicialmente, ou seja, ao ser criado um cartão este possui

[1] Este artigo encontra-se também publicado na Internet no site http://www.cs.auc.dk/~csj/Glossary

um tempo associado desde a data de criação até a data de vencimento. Esta data de vencimento, ao ser atingida, pode ser estendida por mais cinco anos e o tempo de validade do cartão é então alterado e aumentado para conter esta nova validade e assim por diante ou até que o dono do cartão não mais o queira. Pode-se então modelar a relação "Cartão de Crédito" no banco de dados contendo uma dimensão temporal de tempo de validade que irá conter a validade do cartão.

O tempo de validade pode ser alterado a qualquer momento no banco de dados com seus tempos podendo se referir ao presente, passado e até ao futuro como é o caso do exemplo do cartão de crédito. Como exemplo de uso do tempo de validade no passado podemos utilizar um exemplo de contas bancárias. Normalmente, é importante para um cliente de um banco possuir conta corrente há muito tempo. Na verdade, quanto mais antigo é o cliente, mais prestígio ele tem ao efetuar as suas compras e é lógico que o tipo de conta (comum, especial, etc.) que ele possui tem influência ainda maior em seu prestígio. Imagine que um determinado banco, para atrair clientes de outros bancos, faça uma promoção que, entre outras facilidades, cada cliente que vier de outro banco terá sua data de adesão modificada para sua data de adesão em seu banco anterior. A relação conta corrente possui um atributo de tempo de validade, neste caso, e quando um cliente é cadastrado no sistema, a data de adesão pode ser alterada por um valor no passado.

2.1.2. Relação de Tempo de Validade

"Uma relação de tempo de validade é uma relação com um e somente um tempo de validade suportado pelo sistema. Não há nenhuma restrição sobre como o tempo de validade é incorporado às tuplas da relação."

Uma relação de tempo de validade é uma relação temporal do banco de dados com o tempo de validade incorporado as tuplas. Quando se diz que o tempo de validade é suportado pelo sistema tenta-se dizer que o sistema deve saber lidar com a informação temporal, suportar consultas temporais, de preferência de forma eficiente.

2.1.3. Tempo de Transação

"Um fato é armazenado em um banco de dados em algum ponto no tempo, e após ser armazenado, ele é presente até que seja logicamente removido. O tempo de

transação de um fato em um banco de dados é o tempo onde o fato é presente no banco de dados e pode ser recuperado".

O tempo de transação é referente ao tempo em que os fatos vão sendo armazenados, alterados ou logicamente removidos no banco de dados. Os fatos num banco de dados temporal nunca são fisicamente removidos, mas sim logicamente removidos, ou seja, permanecem armazenados no banco de dados apenas com informação de que foram removidos para que consultas a dados já removidos possam se tornar possíveis.

Iremos chamar o tempo em que o fato é inserido no banco de dados de tempo de nascimento (*t_nasc*) do fato e o tempo em que é logicamente removido de tempo de morte (*t_morte*) do fato. O tempo de transação pode ser visto então como um intervalo aberto de tempo de vida transacional do fato indo desde o seu tempo de nascimento até seu tempo de morte e é representado da seguinte forma [*t_nasc*,*t_morte*). Ao ser inserido o fato no banco de dados este intervalo recebe o valor [*t_nasc*,*), onde "*" representa o tempo atual ou tempo corrente e significa que o fato encontra-se vivo no banco de dados, ou seja, ainda não foi removido.

O tempo de transação, ao contrário do tempo de validade, pela natureza transacional do mesmo, só pode ser alterado no tempo atual do banco de dados, ou seja, alterações só podem ser feitas em objetos vivos (com *t_morte*=*) e no tempo atual.

2.1.4. Relação de Tempo de Transação

"Uma relação de tempo de transação é uma relação com um e somente um tempo de transação suportado pelo sistema. Assim como para relações de tempo de validade não há nenhuma restrição sobre como o tempo de transação é incorporado às tuplas da relação."

Uma relação de tempo de transação funciona da mesma maneira que uma relação de tempo de validade. A diferença fica por conta das diferenças dos tempos de validade e transação. Aqui, o tempo não é explicitamente manuseado pelo usuário e sim são tempos gerados pelos controles de transações do sistema. O usuário não tem o controle sobre os tempos de transação apenas efetuando as operações básicas de

inserção, remoção e alteração dos objetos. O sistema altera os tempos de vida transacional dos objetos conforme a operação sendo efetuada.

2.1.5. Relação Bitemporal

"Uma relação bitemporal é uma relação com um e somente um tempo de transação suportado pelo sistema e com um e somente um tempo de validade suportado pelo sistema. Este tipo de relação herda as propriedades das relações de tempo de transação e de validade. Não há nenhuma restrição sobre como ambas as informações temporais são incorporadas às tuplas da relação."

2.1.6. Instante de Tempo

"*Um instante de tempo é um ponto no eixo dos tempos*".

O conceito de instante de tempo serve tanto para tempo de validade quanto de transação. É importante salientar que a noção de tempo pode ser visto como uma variável contínua ou discreta. Nesta dissertação só estaremos lidando com o caso do tempo como uma variável discreta, logo um instante de tempo é um ponto no eixo dos tempos discretizado.

2.1.7. Intervalo de Tempo

"*Um intervalo de tempo é o tempo entre dois instantes*".

Um intervalo de tempo entre os tempos t_1 e t_2 pode ser definido como o intervalo aberto entre os dois tempos e representado da seguinte forma: $[t_1, t_2)$.

2.1.8. Banco de Dados Temporal

Finalizando, um banco de dados pode ser dito temporal (BDT) se possui suporte para o armazenamento e recuperação de informações temporais. Pelo armazenamento de informação temporal entende-se que um BDT deve possuir suporte para relações de tempo de transação, de validade ou relações bitemporais. Por recuperação de informação temporal, um BDT deve possuir uma linguagem de consulta declarativa,

mecanismos de indexação eficientes para dar suporte a estas consultas e por fim (fora do escopo desta dissertação) um mecanismo de otimização destas consultas.

2.2. Consultas Temporais

Várias propostas de linguagens de consultas temporais existem na literatura atualmente. Dentre elas podemos ressaltar dois importantes trabalhos: TQuel (SNODGRASS, 1987) e TSQL2 (SNODGRASS, 1995). Uma linguagem de consulta temporal pode consultar dados no passado, presente e no futuro (apenas para tempo de validade). A principal variação em uma consulta temporal é se o argumento temporal é um instante de tempo ou um intervalo de tempo. Esta última consulta por intervalo de tempo, segundo LANGRAN (1992), é uma das consultas mais importantes que um Sistema de Informações Geográficas Temporal deve responder. São exemplos de consultas temporais:

- Instante de tempo: "Retornar todos os empregados que trabalhavam nesta empresa no ano de 1999";
- Intervalo de tempo: "Retornar todos os empregados que trabalharam nesta empresa durante o período contido entre os anos de 1995 a 1999";
- Intervalo de tempo (tempo de validade no futuro): "Retornar todos os clientes cuja validade do cartão irá expirar entre maio/2002 e dezembro/2002".

Outro tipo importante de consulta temporal é a junção temporal (GUNADHI et al., 1991). A junção temporal utiliza as informações contidas nas chaves das tuplas bem como a informação temporal da mesma. A Junção-T (*T-Join*) recebe como parâmetros duas relações temporais e retorna um conjunto de pares de tuplas destas relações cuja informação temporal possui interseção; a Junção-TE (*TE-Join*) recebe como parâmetros também duas relações temporais retorna um conjunto de pares de tuplas cujas chaves são iguais e a informação temporal possui interseção. Em ZHANG (2000) é dito que este segundo tipo de junção (*TE-Join*) é o mais comumente utilizado, logo o mais importante que um BDT deve suportar eficientemente.

2.3. Consultas Espaciais

Bancos de Dados Espaciais (BDEs) permitem o armazenamento de dados espaciais como pontos, linhas, polígonos, etc. Aplicações como Sistemas de Informações Geográficas (SIG), Projetos de VLSI (do inglês, *Very Large Scale Integration*), Sistemas Multimídia, entre outros, têm se beneficiado bastante do uso de BDEs ou de técnicas empregadas nos mesmos.

Dados espaciais podem ser facilmente armazenados em bancos de dados convencionais projetados pelos usuários, como por exemplo para representar uma cidade armazenar seu nome e suas coordenadas como dois valores representando latitude e longitude. O problema desta representação encontra-se na dificuldade de se representar os relacionamentos espaciais entre os objetos. As relações espaciais entre os objetos podem ser topológicas (interseção, fronteira, etc.) ou direcionais (ao norte, ao sul, etc.). Estas relações entre objetos espaciais encontram-se melhores descritas em PAPADIAS et al. (1997). A representação espacial dos objetos para que a recuperação dos dados seja eficiente deve ser parte integrante e interna do banco de dados. Mecanismos de indexação dos objetos espaciais devem ser providos por um BDE a fim de permitir que as consultas espaciais sejam eficientes e em alguns casos extremos até factíveis.

Segundo BRINKHOFF et al. (1993), as consultas espaciais se dividem em dois grandes grupos: consultas de varredura simples (*single-scan queries*) e consultas de varredura múltipla (*multiple-scan queries*). As consultas de varredura simples requerem no máximo um acesso por cada objeto espacial da relação e portanto seu tempo de execução é uma função linear do número de objetos existentes na correspondente relação. As consultas de varredura múltipla, por sua vez, requerem que cada objeto seja acessado mais de uma vez, não possuindo tempo de execução linear, mas superlinear com relação ao número de objetos da relação. Um exemplo de uma consulta de varredura simples é a consulta por janela (*window query*) que, dada uma determinada janela R (um retângulo alinhado com os eixos), retorna todos os objetos de uma relação que possuem interseção com R. Um exemplo de consulta por varredura múltipla é a junção espacial (*spatial join*) (ORENSTEIN, 1986) e a operação de sobreposição de mapas (*map overlay*) (BURROUGH , 1986). A junção espacial é a operação que

combina duas relações segundo alguma propriedade espacial, como por exemplo a interseção. A sobreposição de mapas é a operação utilizada para sobrepor dois ou mais mapas gerando um mapa final com todas as informações pertinentes. A junção espacial e a sobreposição de mapas podem ser vistas na Figura 1 abaixo.

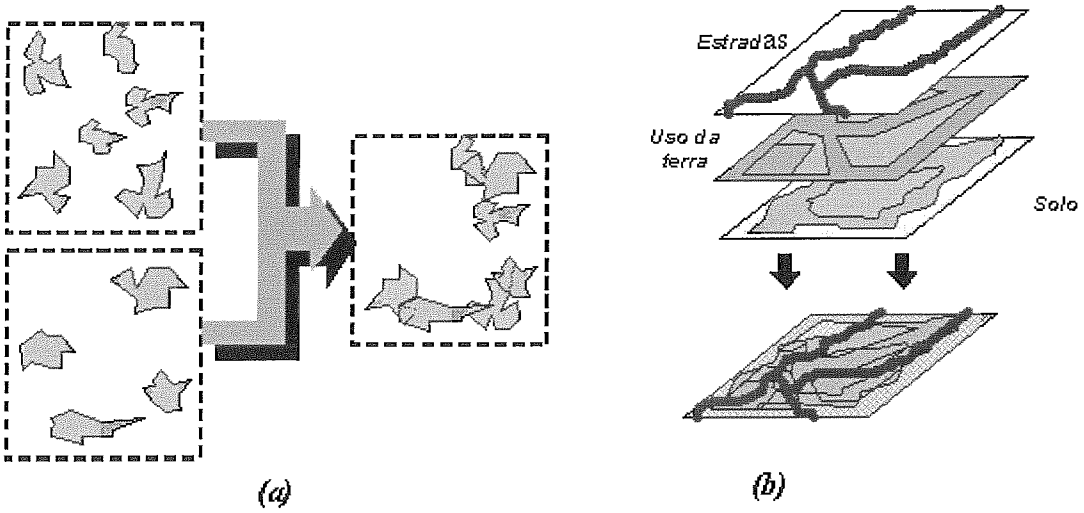


Figura 1. *Exemplos de (a) junção espacial e (b) sobreposição de mapas*

Uma descrição formal da junção espacial e o processamento eficiente utilizando-se R*-Trees pode ser vista em BRINKHOFF et al. (1993). Cinco algoritmos são propostos e uma análise de desempenho destes algoritmos é apresentada e bastante bem discutida neste artigo.

2.4. Consultas Espaçotemporais

Juntando as dimensões de tempo e espaço surgem os Bancos de Dados Espaçotemporais que englobam armazenamento e recuperação eficientes de dados espaciais e temporais ao mesmo tempo. Há uma gama de aplicações que fazem uso de dados espaçotemporais e portanto necessitam da utilização de BDET, como por exemplo sistemas multimídia (animações), sistemas de monitoração remota por satélite, sistemas de informações geográficas, a computação móvel, etc. Em todas estas aplicações os objetos espaciais se movem e mudam de formato ao longo do tempo e estas modificações devem se manter armazenadas.

É importante frisar que em aplicações espaciais, o volume e a complexidade dos dados utilizados são normalmente maiores que os das aplicações convencionais e, como em bases de dados espaçotemporais os objetos nunca são fisicamente removidos, este

volume cresce ainda mais para aplicações espaçotemporais. Estruturas de indexação dos dados espaçotemporais são ainda mais necessárias para a recuperação eficiente dos objetos.

As consultas espaçotemporais especificam predicados de consulta espaciais e temporais de forma combinada e os objetos que as satisfazem são retornados. Podemos ter, portanto, consultas por janela por instante de tempo, consultas por janela por intervalo de tempo, junções espaçotemporais, etc.

As consultas espaçotemporais podem ser separadas em três principais categorias de acordo com a nomenclatura em LANGRAN (1992): (a) as consultas predominantemente espaciais, (b) as consultas predominantemente temporais e (c) as consultas sem predominância de tempo ou espaço. Um coeficiente de predominância espaçotemporal (θ) encontra-se definido na Eq. 1 para identificar a classificação das consultas. Este coeficiente é a razão da parcela ocupada pela parte espacial da consulta pela parcela ocupada pela parte temporal da consulta. Quanto maior for este coeficiente maior a predominância espacial e quanto menor, maior a predominância temporal. A Figura 2 mostra graficamente a predominância espaçotemporal segundo o crescimento do coeficiente θ.

$$\theta = \frac{área\_janela / área\_total}{\left(tamanho\_intervalo / tempo\_total \right)^2} \qquad \text{(Eq. 1)}$$

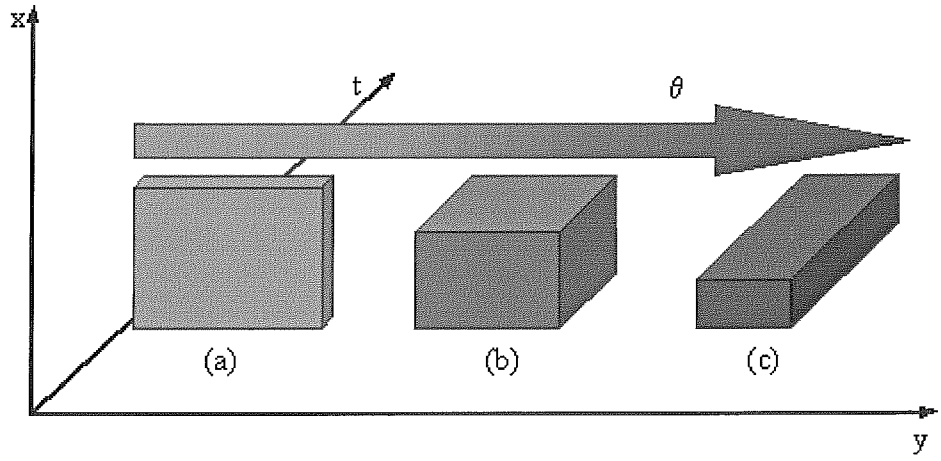


Figura 2. Gráfico de predominância das consultas. (a)Consultas predominantemente espaciais (b)Consultas sem predominância (c)Consultas predominantemente temporais.

Por fim, seguem as junções espaçotemporais. A junção espaçotemporal nada mais é do que a execução de uma junção espacial tomando como argumentos um intervalo de tempo ou um instante de tempo.

A junção espaçotemporal por instante de tempo retorna os mesmos resultados da junção espacial das relações como elas eram no instante de tempo da consulta, ou seja, retorna todos os pares de objetos que satisfazem a condição de junção (interseção, proximidade, etc.) entre si e que estavam vivos no instante de tempo da consulta.

A junção espaçotemporal por intervalo de tempo torna-se ainda mais complexa pois irá retornar todos os pares de objetos espaciais que satisfazem a condição de junção entre si e encontravam-se vivos durante o intervalo de tempo determinado.

2.5. Sumário

Neste capítulo procuramos definir alguns termos de bancos de dados espaciais e temporais que serão utilizados ao longo da dissertação. Procuramos principalmente definir o que entende-se por consultas temporais, espaciais e espaçotemporais, que é o objeto de estudo desta dissertação.

No próximo capítulo será feita uma revisão da bibliografia sobre métodos de acesso temporais, espaciais e espaçotemporais.

3. Índices Espaçotemporais

Neste capítulo iremos mostrar como as consultas temporais, espaciais e principalmente espaçotemporais podem ser auxiliadas com a existência de métodos de acesso eficiente aos objetos. Iremos fazer uma revisão da literatura de métodos de acesso espaciais e temporais para facilitar o entendimento dos métodos de acesso espaçotemporais que na maioria das vezes une estas duas técnicas. Cabe notar que não estaremos fazendo uma revisão global da bibliografia de métodos de acesso temporais, espaciais e espaçotemporais, mas sim uma revisão dos principais métodos que dizem respeito ao conseqüente desdobramento desta dissertação.

3.1. Métodos de Acesso Espaciais

3.1.1. R-Tree

A R-Tree foi proposta por GUTTMAN (1984) e foi um marco no desenvolvimento de estruturas de índice espaciais. Esta estrutura é uma árvore balanceada baseada nos princípios da B+-Tree (COMER, 1979) para a indexação de objetos espaciais. Os objetos espaciais (polígonos, pontos, linhas, etc.) são representados internamente na estrutura através de seus Mínimos Retângulos Envolventes (MBRs, do inglês Mininum Bounding Rectangles). Um MBR é a representação do objeto espacial em um retângulo (alinhado aos eixos) que envolve completamente o mesmo. Cada entrada em um nó interno contém um conjunto de entradas contendo ponteiro para nós filhos na árvore e um MBR que engloba totalmente os MBRs destes nós filhos. Da mesma forma, um nó folha contém um conjunto de registros de índice contendo os MBRs dos objetos espaciais indexados na estrutura e seus respectivos ponteiros, onde possam ser encontrados.

Seja então M o número máximo de entradas em um nó e $m \leq M / 2$ um parâmetro identificando o número mínimo de entradas em um nó, seis propriedades são propostas em GUTTMAN (1984) como descrito a seguir:

R1. Cada nó folha contém entre *m* e *M* registros de índice ao menos que seja um nó raiz;

R2. Para cada registro de índice < *MBR*, *ponteiro_objeto_espacial* > em um nó folha, MBR é o menor retângulo paralelo aos eixos que espacialmente contém o objeto espacial;

R3. Cada nó interno contém entre *m* e *M* nós filhos ao menos que seja um nó raiz;

R4. Para cada entrada < *MBR*, ponteiro_filho > em um nó interno, MBR é o menor retângulo que espacialmente contém todos os MBRs do nó filho;

R5. O nó raiz tem pelo menos dois filhos ao menos que seja uma folha;

R6. Todas as folhas aparecem no mesmo nível da árvore.

Essas características, bem como na B+-Tree, que garantem uma boa eficiência assintótica de tempo de consultas bem como de utilização de espaço. É mostrado em GUTTMAN (1984) que uma árvore contendo N registros de índice com essas características acima descritas possui altura de no máximo $\lceil \log_m N \rceil - 1$. Um exemplo de uma R-Tree pode ser vista na Figura 3 abaixo retirada de GUTTMAN (1984).

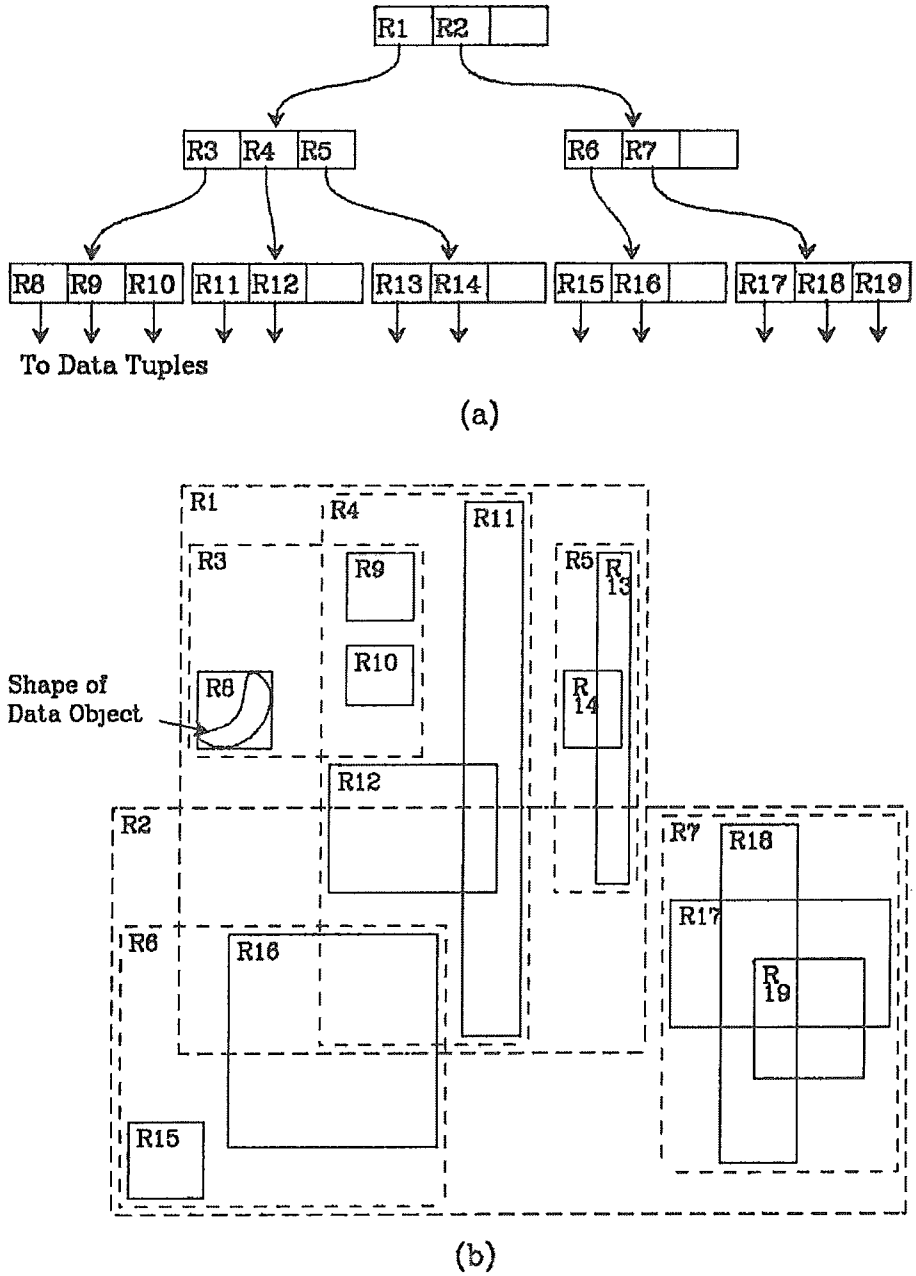


Figura 3. ***Um exemplo de uma R-Tree, sua representação estrutural em (a) e sua representação espacial em (b)***

Podemos notar nesta figura como os objetos são envolvidos pelos seus respectivos MBRs. Podemos também perceber como é feito o agrupamento de objetos em cada nó da estrutura, levando-se em conta suas posições espaciais, ou seja, podemos perceber como funciona a idéia principal deste método de acesso espacial, que é a de manter no mesmo nó objetos que se encontram espacialmente próximos uns dos outros.

O algoritmo de busca é bastante simples e similar ao da B+-Tree descendo-se na árvore a partir do nó raiz até as folhas por caminhos que obedecem à restrição da busca. Diferentemente da B+-Tree, mais de um caminho pode obedecer à restrição da busca

devido à inexistência de uma linearização do espaço e conseqüentemente sobreposição de MBRs. Perceba ainda na Figura 3 a sobreposição existente entre os nós R1 e R2. Com isso não se pode garantir uma boa complexidade de pior caso para o algoritmo de busca. Entretanto, a forma com que os objetos são inseridos nos índices, principalmente a forma com que se faz a divisão dos nós com overflow, é um fator crucial para a complexidade de caso médio.

O algoritmo de inserção, assim como o de busca, é semelhante ao da B+-Tree. Inicialmente é encontrado o nó folha onde o MBR do objeto será inserido. Este nó é encontrado descendo-se na árvore de forma semelhante ao algoritmo de busca procurando sempre o nó que terá um menor aumento de área caso recebesse o MBR do objeto. Caso haja espaço para o novo objeto, ele é inserido e o algoritmo acaba. Caso contrário, o nó é dividido em dois e a divisão é propagada até o nó raiz. Três algoritmos de divisão de nós são apresentados nesse trabalho e nomeados segundo sua complexidade: o algoritmo exaustivo, o algoritmo de complexidade quadrática e o algoritmo de complexidade linear. Todos os três os algoritmos procuram minimizar a área dos dois MBRs dos nós obtidos após a divisão.

O algoritmo exaustivo de divisão dos nós testa todas as possibilidades possíveis verificando ao final qual organização possui menor área. É demonstrado que sua complexidade é inviável para a utilização no algoritmo de inserção. O método quadrático inicia escolhendo-se duas sementes como sendo os dois objetos mais distantes um do outro e colocando cada um em um nó. Após este passo, segue escolhendo o objeto com mais afinidade por um dos dois nós e inserindo-o no mesmo. O algoritmo pára quando acabam os objetos a dividir ou quando a quantidade de entradas em um dos dois nós acrescida do resto dos objetos restantes é igual a m, ou seja, todas as entradas restantes devem ser inseridas neste nó ou então a propriedade do número mínimo de entradas seria violada. O algoritmo linear utiliza um processo semelhante de escolha das sementes que apenas verifica a distância em um dos eixos. O próximo passo do algoritmo também não busca encontrar o objeto com mais afinidade por um dos nós e simplesmente percorre a lista de objetos restantes sem ordenação escolhendo o nó de maior afinidade. É mostrado neste artigo que o algoritmo que dá melhores resultados é o de divisão quadrática de nós.

O algoritmo de remoção inicia com uma busca para a obtenção da entrada a ser removida. Uma vez encontrada, a entrada é removida do nó folha onde se encontra. Se este nó violar a regra de número mínimo de entradas, todas as entradas são removidas e inseridas em um conjunto de entradas a serem reinseridas posteriormente. Esta remoção pode ser propagada até o nó raiz. Após esse processo, todas as entradas do conjunto de entradas removidas são reinseridas na árvore a partir do nó raiz usando-se um algoritmo semelhante ao de inserção. Este algoritmo tem que ter o cuidado de inserir as entradas exatamente nas alturas em que elas pertenciam quando foram removidas da árvore. Ao final da reinserção de entradas removidas, a condição de número mínimo de entradas no nó raiz é testada e, uma vez violada (garantidamente o nó raiz encontra-se com apenas uma entrada), o algoritmo troca o nó raiz pelo nó apontado por sua única entrada.

Após o surgimento da R-Tree, muitas estruturas foram propostas baseadas em seus princípios com melhorias de desempenho, dentre elas podemos citar a R+-Tree (SELLIS et al. 1987), a R*-Tree (BECKMANN et al., 1990) e a Hilbert R-Tree (KAMEL et al., 1994).

3.1.2. R*-Tree

A R*-Tree foi proposta por BECKMANN et al. (1990) com o objetivo principal de otimizar os algoritmos propostos na R-Tree original de Guttman. O foco principal das otimizações baseia-se no fato de que a heurística proposta na R-Tree original de minimizar a área dos MBRs formados foi amplamente utilizada mas não foi mostrada ser a melhor heurística, nem ao menos foram testadas outras heurísticas. Observações contundentes bem como uma grande massa de testes foi utilizada neste trabalho para basear suas conclusões.

Existem quatro parâmetros essenciais que afetam o desempenho da estrutura e ainda, os parâmetros não são interdependentes:

(O1) A área coberta por um *MBR* envolvente mas não coberta pelos *MBR*s envolvidos (área morta) deve ser minimizada;

(O2) A área de interseção entre MBRs deve ser minimizada;

(O3) O perímetro dos MBRs deve ser minimizado;

(O4) O espaço de armazenamento deve ser otimizado.

A R-Tree só utiliza o parâmetro de otimização **O1** em seus algoritmos e conseqüentemente **O4**, por construção. Aí se encontra a grande falha desta estrutura, ela não leva em conta outros parâmetros também essenciais para a otimização de desempenho.

A R*-Tree propõe então uma alteração nos algoritmos de escolha do nó onde encaixar a entrada a ser inserida (*ChooseSubtree*) e no algoritmo de divisão de nós (*Split*).

O algoritmo *ChooseSubtree* segue os mesmos princípios (parâmetro de otimização **O1**) da R-Tree original enquanto percorre nós internos que não apontam para folhas, e no momento em que chega em nós que apontam para folhas o parâmetro utilizado é o de minimizar a área de interseção (**O2**). Ainda existe uma otimização para esse algoritmo para reduzir o custo de utilização de CPU e pode ser melhor visualizado em BECKMANN et al. (1990).

Quanto ao algoritmo de divisão de nós, é proposto um algoritmo totalmente diferente dos três propostos na R-Tree (exaustivo, quadrático e linear). O algoritmo inicia separando por eixo os retângulos em dois conjuntos ordenados por valor mínimo (m) e por valor máximo (M) de sua coordenada no eixo determinado. Para cada conjunto podemos obter $M-2m+2$ distribuições das M+1 entradas em dois grupos como pode ser visto na Tabela 1 abaixo.

Tabela 1. *Distribuição de entradas pelos conjuntos de divisão de nós na* ***R*-Tree***

Valores de k	Primeiro grupo (m-1)+k entradas	Segundo grupo entradas restantes
1	*m*	*M+1-m*
2	*m+1*	*M-m*
M	*M*	*M*
M-2m+2	*M+1-m*	*m*

Para cada distribuição, três valores, cujos nomes encontram-se em sua versão original na língua inglesa devido à utilização dos mesmos nos algoritmos, são determinados para se escolher a melhor distribuição final dos nós:

(i) *area-value*: área[MBR(primeiro grupo) + área[MBR(segundo grupo)

(ii) *margin-value*: perímetro[MBR(primeiro grupo) + perímetro[MBR(segundo grupo)

(iii) *overlap-value*: área[MBR(primeiro grupo) $\cap$ MBR(segundo grupo)]

Dados os *M-2m+2* conjuntos, o eixo de divisão é determinado utilizando-se o parâmetro **O3** que minimiza o perímetro dos MBRs dos grupos. Uma vez escolhido o eixo, o melhor grupo é escolhido primeiro pelo parâmetro **O2** e depois por **O3**.

Uma outra otimização proposta na R*-Tree é a reinserção forçada. Foi verificado um comportamento não determinístico nessas estruturas, isto é, diferentes ordens de inserção levam a diferentes estruturas construídas. Isto gera o problema de uma árvore deteriorada ao longo do tempo de utilização. Um teste interessante foi feito com a R-Tree utilizando-se o algoritmo de divisão de nós linear que mostra exatamente esse problema. Foram inseridos 20.000 retângulos uniformemente distribuídos e depois 10.000 deles foram removidos e reinseridos novamente. O resultado foi um ganho de performance de 20 até 50% dependendo do tipo de consulta. Para obter então uma reorganização dinâmica da árvore, sem que de tempos em tempos tenha que ser feita uma reorganização geral foi proposto um algoritmo de reinserção forçada das entradas embutido no algoritmo de inserção. Quando ocorre um overflow de um nó, ao invés de partir direto para executar uma divisão do mesmo, o algoritmo de inserção remove um percentual de entradas deste nó recolocando-o nas condições de valores mínimo e

máximo de entradas. Estas entradas são reinseridas utilizando-se o mesmo algoritmo de reinserção de entradas do algoritmo de remoção.

3.1.3. Hilbert R-Tree

A Hilbert R-Tree é mais uma variante da R-Tree que obteve um desempenho muito bom. Foi proposta em 1994 por KAMEL et al. (1994) e a idéia principal dessa estrutura é a de manter uma ordenação nos nós da árvore para que se possa adiar a divisão de nós assim como é feito nas B+-Trees. Esta ordenação linear é obtida através de uma curva de preenchimento do espaço, a curva de Hilbert, onde cada retângulo tem a coordenada de seu centro mapeada por um valor, o valor de Hilbert. Um exemplo desta curva pode ser vista na Figura 4 abaixo retirada de KAMEL et al. (1994).

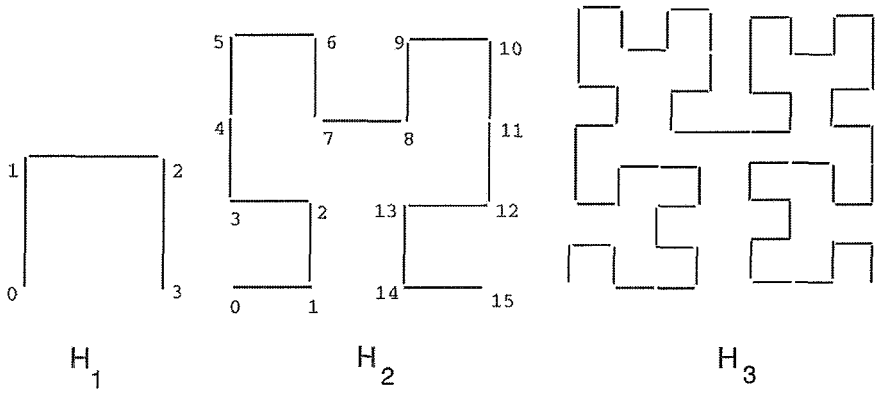


Figura 4. *Curvas de Hilbert de ordens 1, 2 e 3*

Desta forma, a Hilbert R-Tree funciona exatamente como uma R-Tree nos algoritmos de busca e exatamente como uma B+-Tree no algoritmo de inserção utilizando o valor de Hilbert como chave. Cada nó interno *n* na árvore possui então mais um campo com um valor de Hilbert (LHV – do inglês, *Largest Hilbert Value*) que representa o maior valor de Hilbert de todos os retângulos que pertencem à sub-árvore cujo nó raiz é *n*. Um exemplo da estrutura de uma Hilbert R-Tree e do mapeamento dos centros dos MBRs em LHVs pode ser visto na Figura 5 retirada de KAMEL et al. (1994).

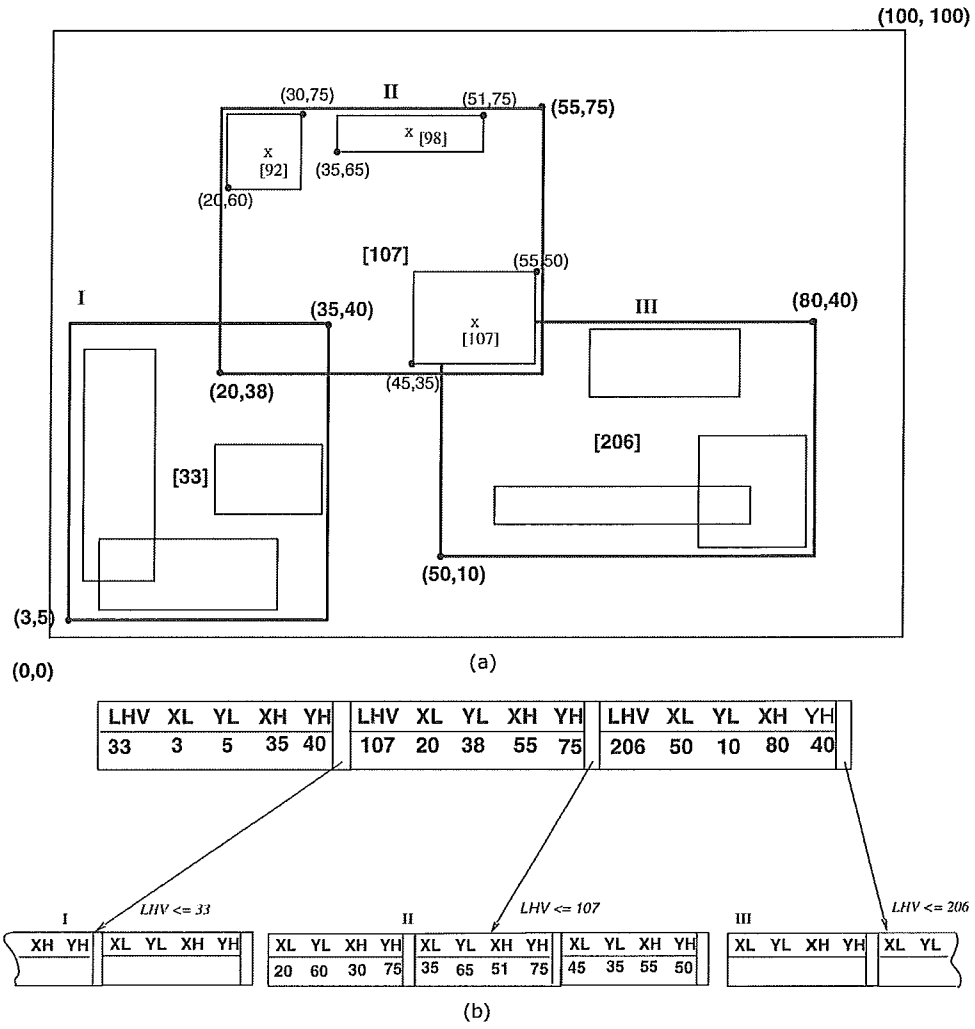


Figura 5. ***Exemplo de uma Hilbert R-Tree. (a) Estrutura espacial dos MBRs e seus respectivos LHVs. (b) Estrutura física da árvore para os MBRs em (a).***

O algoritmo de inserção inicialmente procura o nó onde o novo retângulo será inserido através de seu valor de Hilbert. Se este nó estiver cheio, o algoritmo tenta distribuir as entradas deste nó cheio com (s – 1) vizinhos do mesmo o que pode gerar, em caso da divisão ser inevitável, (s + 1) nós. Este parâmetro *s* é determinante e diretamente proporcional ao percentual de utilização de espaço da estrutura. Esta política de divisão dos nós é chamada de 1-para-2, 2-para-3, 3-para-4, ... conforme o valor de *s* = 1, 2, 3, ... É mostrado neste artigo que o valor ideal para *s* é 2, ou seja, a política de 2-para-3 é a ideal.

A remoção na estrutura é feita também de forma semelhante à B+-Tree. Quando ocorre um underflow em um nó, pega-se algumas entradas de seus nós vizinhos ou então junta-se os s+1 vizinhos para formar s nós. É importante salientar que nunca ocorre reinserção de nós devido à ordenação proposta.

3.2. Métodos de Acesso Temporais

Esta seção será dividida em métodos de acesso que suportam somente o tempo de transação e métodos de acesso bitemporais (suportam tempo de transação e de validade simultaneamente). Não existem métodos de acesso que suportam somente o tempo de validade pois devido à natureza do tempo de validade, métodos de acesso por intervalo, como por exemplo a Segment R-Tree ou simplesmente SR-Tree (KOLOVSON et al., 1991), encaixam-se perfeitamente para este caso.

3.2.1. Métodos de Acesso com Suporte a Tempo de Transação

3.2.1.1. Overlapping B+-Tree (OBT)

Uma idéia inicial para armazenar estados passados dos dados em B+-Trees seria a de se criar uma nova B+-Tree a cada nova versão dos dados. Isto seria bastante ineficiente e haveria um enorme desperdício de espaço. Para evitar esse problema foi proposta a Overlapping B+-Tree (MANOLOPOULOS et al., 1990) baseada nos princípios de árvores sobrepostas expostos em BURTON et al. (1985). Esta estrutura possui uma B+-Tree para cada versão dos dados, os caminhos da árvore que não sofreram alterações entre versões são compartilhados. Observe na Figura 6 a alteração de um registro contido no nó folha hachurado da árvore e como fica a estrutura após esta alteração. Note que todo o caminho do nó que sofreu a alteração até o nó raiz da árvore atual é duplicado e os outros permanecem compartilhados.

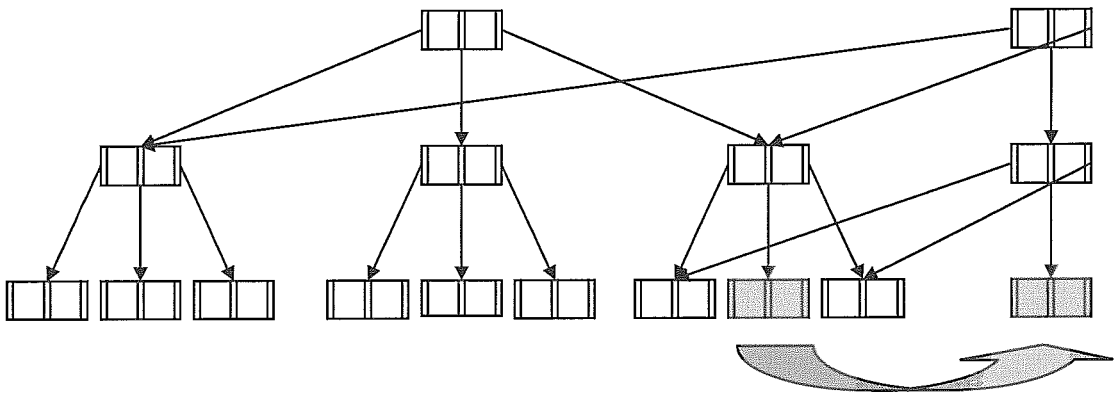


Figura 6. Estrutura de uma Overlapping B+-Tree

Cada nó raiz deve possuir um atributo que represente seu tempo de criação para que cada versão possa ser consultada. Os nós raiz ficam em uma outra estrutura que

pode ser uma tabela de hash ou até uma outra B+-Tree indexada por esse tempo de criação dos nós.

A grande vantagem desta estrutura é a simplicidade de implementação uma vez que é bastante semelhante a uma B+-Tree. Esta estrutura possui bons tempos de resposta para as consultas por instante de tempo e um desempenho ruim para consultas por intervalo de tempo, pois pode ser necessário percorrer mais de uma B+-Tree durante a consulta. Isto se agrava ainda mais com o aumento do intervalo de tempo, pois torna-se necessária uma busca por uma grande quantidade de B+-Trees. Uma outra desvantagem é que algum mecanismo de eliminação de duplicatas deve ser acoplado ao resultado das consultas por intervalo de tempo. Uma outra característica não muito boa desta estrutura é o aproveitamento de espaço, uma vez que há muita duplicação de nós caso existam muitas versões na base de dados.

3.2.1.2. Multiversion B-Tree (MVBT)

Esta estrutura foi apresentada em OHLER (1994) e BECKER et al. (1996) como uma solução para o armazenamento assintoticamente eficiente de dados contendo diversas versões. A Multiversion B+-Tree ou MVBT consegue manter as mesmas boas propriedades da B+-Tree, no pior caso.

A MVBT também é uma estrutura de sobreposição de árvores, assim como o método exposto acima, a OBT. Diferentemente da OBT, a MVBT armazena em um mesmo nó objetos de tempos de vida diferentes e por isso torna-se necessária que essa informação esteja contida nas entradas dos nós. Uma entrada de um nó fica então da forma <*chave, referência, t_nasc, t_morte*> onde para os nós internos *referência* aponta para o nó filho na árvore e para os nós folha *referência* aponta para o verdadeiro objeto externamente ou é a própria informação referente ao objeto.

As boas propriedades são garantidas por duas condições: a condição fraca de versão (*weak version condition*) e a condição forte de versão (*strong version condition*).

Seja N a versão corrente da estrutura e b a capacidade de um nó, a condição fraca de versão é similar à condição de armazenamento da B+-Tree e assegura que:

- o número de entradas vivas na versão corrente N em um nó raiz deve ser no mínimo 2 a menos que este seja um nó folha;

- para um nó não raiz, o número de entradas vivas na versão corrente N é no mínimo $d = k \times b$, para uma constante de ajuste k que no caso de B+-Trees é igual a 0,5.

Esta propriedade deve sempre ser garantida. A violação desta propriedade ocorre quando de uma inserção em um nó cheio (overflow do nó) ou quando de uma remoção restando menos de *d* entradas vivas na versão corrente (underflow fraco do nó). Note que um underflow de nó nunca ocorre desde que uma entrada nunca é fisicamente removida do nó. Assim, uma mudança estrutural é disparada e executada baseada na operação de cópia de nós, isto é, uma divisão de versão (*version split*). A divisão de versão copia para um novo nó todas as entradas vivas do nó na versão corrente e pode ser vista no exemplo da Figura 7.

Após a divisão de versão ocasionada por um overflow de nó, pode ocorrer a situação indesejável de um nó praticamente cheio (ou até cheio) em que próximas inserções dispararam logo novas mudanças estruturais, ou até o caso inverso de um nó praticamente vazio. Para evitar este problema existe a condição forte de versão que é verificada sempre após uma operação de divisão de versão e garante que o número de entradas na versão corrente em um nó após uma mudança estrutural varia de $(1 + \varepsilon) \times d$ e $(k - \varepsilon) \times d$, para um constante de ajuste ε. Em outras palavras, garante-se que após uma mudança estrutural em um bloco no mínimo $\varepsilon \times d + 1$ operações de inserção ou remoção podem ser feitas até que uma nova mudança estrutural seja necessária. Se o número de entradas após uma divisão de versão supera esse limite superior dizemos que ocorreu um overflow de condição forte de versão (*strong version overflow*); se este número é menor que o limite inferior dizemos que ocorreu um underflow de condição forte de versão (*strong version underflow*).

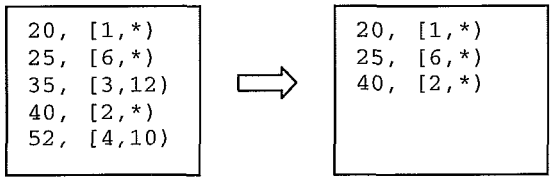


Figura 7. ***Exemplo da divisão de versão de um nó.***

A estrutura da MBVT, assim como a da OBT, não é uma árvore e sim um grafo acíclico direcionado, necessitando de uma estrutura auxiliar contendo os nós raízes das diversas árvores formadoras da estrutura. Cada nó raiz contém, assim como as entradas dos nós, um tempo de vida [*t_nasc*, *t_morte*) e a seqüência de nós raízes formam uma

partição total do tempo do conjunto de dados, não existindo portanto interseção entre os tempos de vida dos nós raízes.

Desta forma, as consultas por instante de tempo podem ser resolvidas da mesma forma como o são nas B+-Trees após ser encontrado o nó raiz que contém o tempo de consulta e levando-se em conta o tempo de vida das entradas nos nós. As consultas por intervalo de tempo não se encontram bem explicitadas neste trabalho mas encontramos uma forma eficiente de percorrer diversos nós raízes neste tipo de estrutura de forma que este tipo de consulta não tenha uma performance ruim assim como nas OBTs. Este método será melhor explicitado no Capítulo 4 quando falarmos da busca por intervalo de tempo da TR-Tree.

Operações de alteração, como inserções e remoções, podem ser executadas também normalmente como em B+-Trees. A diferença encontra-se quando mudanças estruturais tornam-se necessárias.

O algoritmo de inserção inicialmente encontra o nó onde inserir o novo objeto. Se houver espaço a nova entrada é inserida no nó e o algoritmo termina. Caso contrário (overflow de nó) uma divisão de versão é disparada neste nó gerando um novo nó somente com entradas vivas. Deve-se recursivamente inserir uma entrada no nó pai apontando para esse novo nó ligando o novo nó à estrutura. Se neste novo nó encontrar-se em underflow de condição forte de versão, efetua-se uma fusão com um de seus nós vizinhos similar ao da B+-Tree com divisão de versão deste nó vizinho. Em caso contrário, se este nó encontrar-se em overflow de condição forte de versão uma divisão de nós igual a da B+-Tree é efetuada inserindo-se recursivamente uma entrada no nó pai para o segundo novo nó gerado.

Na remoção de objetos pode apenas ocorrer um underflow de condição fraca de versão onde é efetuada uma fusão da mesma forma que na inserção quando ocorre underflow de condição forte de versão.

Ao final deste trabalho fica uma ressalva de que algumas estruturas, dentre elas a R-Tree, podem ser modificadas aplicando-se as mesmas técnicas, por possuírem características semelhantes à B+-Tree. Foi baseado nesta idéia que iniciamos os trabalhos para a construção da TR-Tree que será apresentada no Capítulo 4 a seguir.

3.2.2. Métodos de Acesso Bitemporais

3.2.2.1. Incremental Valid Time Tree

Uma idéia inicial para o armazenamento e recuperação de dados bitemporais apontada em NASCIMENTO et al (1995) seria a de se utilizar uma árvore (B+-Tree) para o armazenamento do tempo de validade, a VTT (*Valid Time Tree*), e outra para o armazenamento do tempo de transação, a TTT (*Transaction Time Tree*). Isto pode ser visto na Figura 8 (a). Este método de armazenamento é chamado de 2LBIT (*Two Level Bitemporal Indexing Tree*). É nítido que a quantidade de informação redundante utilizando-se esta abordagem pode vir a ser muito grande.

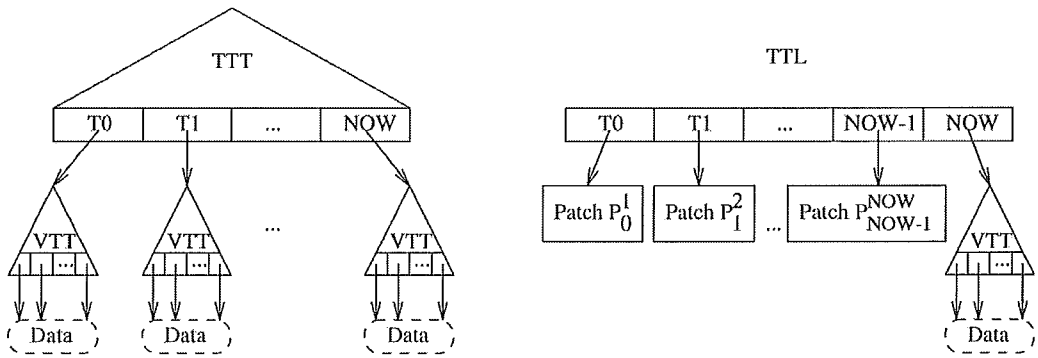


Figura 8. ***Índices Bitemporais (a) Índice bitemporal ingênuo 2LBIT e (b) Incremental Valid Time Tree (IVTT)***

O método IVTT, visto na Figura 8 (b), veio para reduzir a utilização de espaço de armazenamento do 2LBIT dentro da seguinte suposição: "a maioria das consultas referem-se ao estado corrente da base de dados e, para estas, um processamento rápido deve ser garantido. Quanto mais distante no passado uma consulta se refira, menos crítico este tempo de resposta se torna". Dentro desta suposição, a idéia é a de manter apenas a VTT corrente completa e para as outras anteriores, apenas as modificações (*patches*) são armazenadas. Qualquer tempo no passado pode ser reconstituído pegando-se a VTT corrente e aplicando-se os *patches* até este tempo. Desta forma, não há a necessidade de se manter a TTT e sim uma lista com os tempos de transações, a TTL (*Transaction Time List*). Esta técnica garante uma melhor utilização do espaço de armazenamento e um tempo de resposta rápido para consultas em tempos próximos do tempo corrente do banco de dados.

3.2.2.2. Multiple Incremental Valid Time Tree

Esta nova metologia surgiu para resolver o problema do mau comportamento da IVTT para consultas envolvendo tempos no passado longe do tempo corrente do banco de dados. A Multiple Incremental Valid Time Tree (M-IVTT) proposta por NASCIMENTO et al (1996a) permite que várias VTTs completas sejam armazenadas dentro de intervalos de tempos de transação, como pode ser visto na Figura 9. Esta metodologia reduz consideravelmente (e o torna constante) o número de *patches* necessários para se recuperar um estado no passado. A utilização de espaço é um pouco pior que no caso da IVTT, mas não chega a ser tão ruim quanto no caso do 2LBIT.

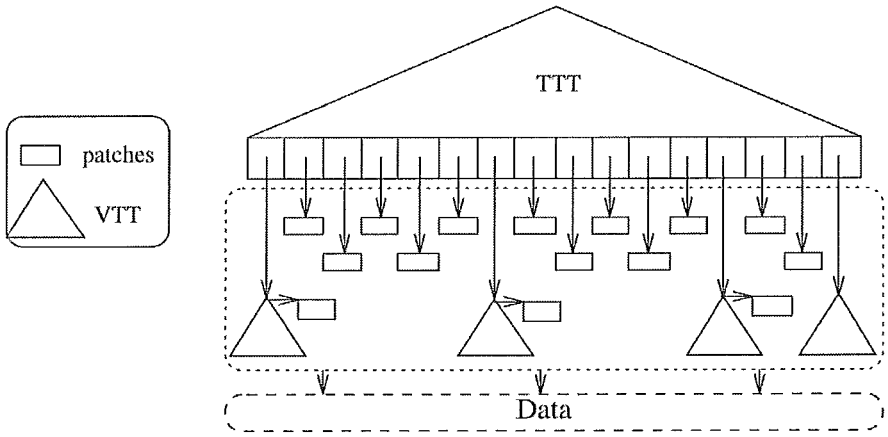


Figura 9. Estrutura da M-IVTT

Estas três abordagens permitem que se faça consultas por intervalo de tempo para o tempo de validade de forma eficiente utilizando-se a técnica de MAP21 (NASCIMENTO et al, 1996b) para armazenar um intervalo de tempo de transação em uma única chave na árvore. A consulta por intervalo de tempo para o tempo de transação não é citada nestes trabalhos.

3.2.2.3. Metodologia de duas R-Trees

Assumir o tempo como uma nova dimensão em uma estrutura de dados espaciais, como a R-Tree, por exemplo, foi uma primeira idéia e parte do princípio que a estrutura de índices utilizada já se encontra implementada. A 3D R-Tree, implementada em THEODORIDIS et al. (1996), é um exemplo de uma estrutura deste tipo no contexto de bancos de dados espaçotemporais. Ela considera o tempo de transação como uma dimensão extra e transforma regiões bidimensionais em tridimensionais. Quando os dados são escalares transformamos o espaço unidimensional

em bidimensional. No contexto de bancos de dados bitemporais, quando considerarmos ainda que o tempo pode ser visto como tempo de transação e de validade estaremos transformando espaços unidimensionais em tridimensionais para dados escalares, por exemplo. A Figura 10 retirada de THEODORIDIS et al (1998) mostra (a) um conjunto de objetos espaciais com seus respectivos tempos de vida e (b) a 3D R-Tree correspondente. Uma busca do tipo "encontre todos os objetos que interceptam uma determinada área D ambos em espaço e tempo" pode ser vista também na figura e é implementada como uma simples busca tridimensional na R-Tree. Esta estrutura é muito boa para o tipo de dados para o qual ela foi proposta (objetos de som e imagem para aplicações multimídia) quando o tempo de vida dos objetos é conhecido. O problema encontra-se quando não se sabe a priori o tempo final dos objetos, ou seja, objetos são criados e permanecem na estrutura até a versão corrente. A R-Tree é uma estrutura muito boa para o armazenamento de MBRs próximos de quadrados (em 2D) e não para retângulos muito compridos, ou seja, para retângulos com muita disparidade entre seus lados, que é o que acontece neste caso do tempo "agora". Uma discussão mais detalhada sobre o problema da versão corrente (tempo "agora") pode ser encontrada em CLIFFORD et al (1997).

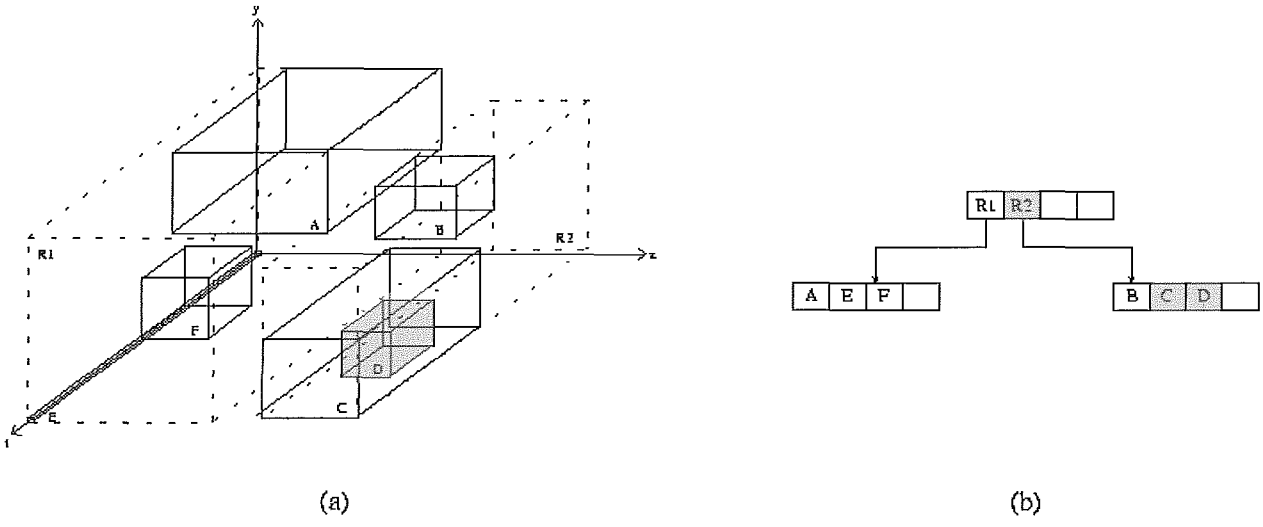


Figura 10. MBRs tri-dimensionais armazenados na 3D R-Tree

Uma forma de resolver este problema do tempo atual é utilizar duas R-Trees como proposto em KUMAR et al. (1995) e em KUMAR et al. (1997) para bancos de dados bitemporais. Na primeira R-Tree ficam armazenados os objetos que estão vivos no tempo atual, armazenando-se na tupla também o tempo de nascimento do objeto, e a segunda funciona como uma 3D R-Tree para os objetos mortos, ou seja, que já possuem

tempo de morte determinado. Quando um objeto é inicialmente inserido na estrutura no tempo t, o mesmo é inserido na R-Tree frontal com $t\_nasc=t$. Quando um objeto é removido no tempo t' ($t'>t$), ele é fisicamente removido da R-Tree 2D e inserido na R-Tree 3D com intervalo de tempo de vida $(t, t']$. Desta forma elimina-se o problema de uma R-Tree armazenar objetos com tempo de vida indeterminado. Um problema adicional criado por esse tipo de estrutura é o de que as buscas por tempo pontual (que não o tempo "agora") e por intervalo de tempo devem prosseguir por duas R-Trees, o que prejudica levemente o desempenho.

Note que esta metodologia não resolve o problema do tempo de morte indeterminado para tempo de validade, apenas para o tempo de transação. Uma suposição feita é a de que o tempo de validade é um intervalo conhecido e que, caso não seja satisfeita, a solução seria o de utilizar um tempo máximo para os tempos de validade indeterminados, o que deteriora o desempenho das R-Trees, pois são construídos retângulos grandes e compridos.

3.2.2.4. Bitemporal R-Tree

A Bitemporal R-Tree (BRT) proposta também em KUMAR et al. (1995) e KUMAR et al. (1997) reduz o problema de bancos de dados bitemporais ao problema de persistência parcial, ou seja, a idéia desta estrutura é a de tornar uma R-Tree bidimensional, com a chave escalar e o tempo de validade como eixos, parcialmente persistente. Segundo BRODAL (1996), uma estrutura de dados parcialmente persistente é uma estrutura em que versões antigas se mantêm armazenadas e podem sempre ser consultadas. Entretanto apenas a última versão ou versão corrente da estrutura de dados pode ser modificada. Devido a estas características, estruturas parcialmente persistentes são perfeitas para o armazenamento do tempo de transação em BDTs. A OBT e a MVBT citadas anteriormente são exemplos de estruturas parcialmente persistentes.

Este método de acesso é uma extensão dos mecanismos propostos em BECKER et al. (1996) para a MVBT para a utilização de uma R-Tree em detrimento ao uso da B+-Tree.

São relatados dois problemas principais para alterações na estrutura da BRT comparando-a à MVBT, principalmente associados à falta de linearização do espaço multidimensional:

- A ordenação das chaves na B-Tree permite que um nó tenha sempre um ou dois vizinhos e estes são utilizados no algoritmo de fusão, caso haja necessidade. Na R-Tree original não existe a noção de fusão exatamente por não existir a noção de nós vizinhos como se fossem próximos espacialmente. Assim, quando ocorre um underflow de um nó, suas entradas são removidas da árvore e reinseridas novamente a partir da raiz. É dito neste trabalho que este método de reinserção não é possível para R-Trees parcialmente persistentes e que a utilização de espaço seria excessiva. Para resolver este problema é proposto então que se faça explicitamente a fusão entre nós vizinhos. Cinco políticas de escolha do vizinho são propostas:

    i. É escolhido o nó vivo filho do mesmo pai com a maior interseção de área de seu MBR com o nó em underflow. Se existirem mais de um, o que terá menor aumento de área é escolhido.

    ii. É escolhido nó vivo filho do mesmo pai que terá menor aumento de área ao incorporar as entradas do nó em underflow.

    iii. É escolhido nó vivo filho do mesmo pai que terá menor aumento de perímetro ao incorporar as entradas do nó em underflow.

    iv. É escolhido nó vivo filho do mesmo pai que terá menor aumento de "área+w*perímetro" ao incorporar as entradas do nó em underflow, onde w>0 é uma constante que define a importância da utilização do perímetro na conta.

    v. É escolhido nó vivo filho do mesmo pai aleatoriamente.

- Quando inserções e remoções não disparam mudanças estruturais na MVBT os objetos são simplesmente inseridos no nó em questão ou removidos do mesmo, nenhum trabalho adicional é efetuado em nós pais deste nó. Em R-Trees, inserções e remoções podem aumentar e diminuir respectivamente o MBR do nó onde objeto está sendo inserido ou removido, o que deve ser propagado pelos pais até o nó raiz. Para evitar o armazenamento desnecessário destas mudanças, a BRT simplesmente ajusta os MBRs dos ancestrais do nó sem que sejam feitas cópias destes nós, neste caso particular.

3.2.2.5. GR-Tree

Em BLIUJUTE et al. (1998) são propostas duas estruturas de índices para bancos de dados bitemporais baseadas em R-Trees que resolvem o problema do tempo crescente para objetos com *t_morte* indeterminado, a GR-Tree e a GR-Tree intermediária. Isto é possível permitindo-se que os MBRs também cresçam ao longo do tempo, tornando-se triangulares ao invés de retangulares para estes casos, o que reduz o espaço de busca consideravelmente. Antes deste trabalho, as estruturas só eram capazes de lidar com o tempo "agora" para tempos de transação, o que pôde ser estendido para o tempo de validade. O tempo "agora" para tempo de transação é chamado de UC (do inglês *Until Changed* que significa até que seja modificado) e para tempo de validade é chamado de NOW (agora).

O aspecto temporal de uma tupla em um banco de dados bitemporal pode ser representado graficamente por uma região bidimensional no espaço com os eixos representando o tempo de transação (eixo das abscissas) e o tempo de validade (eixo das ordenadas). Utilizando essa representação podemos ver na Figura 11 os seis casos possíveis representados na Tabela 2 em que o tempo de transação e/ou tempo de validade de morte dos objetos são indeterminados.

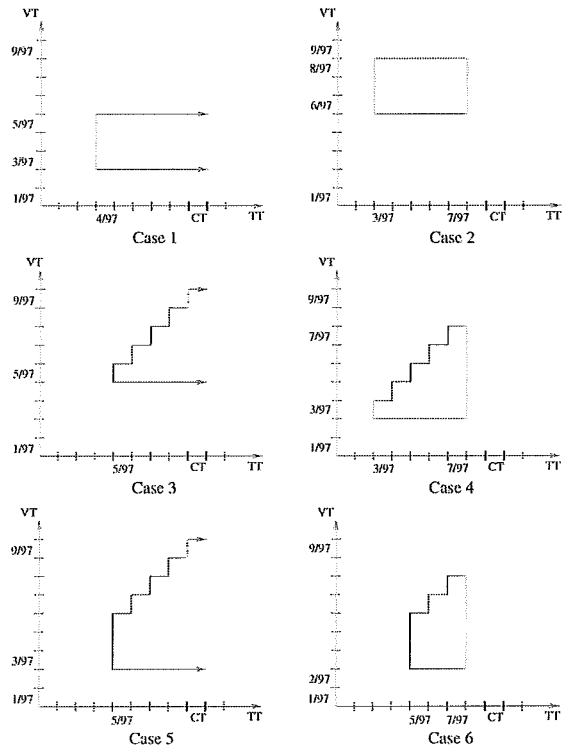


Figura 11. ***Seis possíveis casos para regiões bitemporais com t_morte="agora"***

Tabela 2. *Seis possíveis casos para regiões bitemporais com t_morte="agora"*

	tt_nasc	tt_morte	vt_nasc	vt_morte	
Caso 1	tt1	UC	vt1	vt2	
Caso 2	tt1	tt2	vt1	vt2	
Caso 3	tt1	UC	vt1	NOW	(tt1=vt1)
Caso 4	tt1	tt2	vt1	NOW	(tt1=vt1)
Caso 5	tt1	UC	vt1	NOW	(tt1>vt1)
Caso 6	tt1	tt2	vt1	NOW	(tt1>vt1)

Pode-se notar que quando o tempo de morte de transação dos objetos é conhecido a região é limitada à direita, em caso contrário (UC) cresce indeterminadamente no sentido positivo do eixo das abscissas. Quando o tempo de morte de validade é desconhecido (NOW) é que a região cresce no sentido positivo do eixo das ordenadas. Nestes casos, os MBRs dos objetos tornam-se regiões bitemporais que não representam mais retângulos, mas com formato semelhante ao de uma escada. Estas regiões bitemporais em formato de escada são armazenadas nos nós assim como encontram-se representadas na figura, minimizando o espaço utilizado comparando-se à técnica de utilizar um valor máximo para os tempos indeterminados.

Para a GR-Tree intermediária, estas regiões bitemporais em formato de escada são representadas apenas para os nós folha, com o intuito de diminuir a sobrecarga das operações na estrutura. Para a GR-Tree, os nós internos também incorporam este novo

formato de região envolvente. Uma análise de desempenho é feita ao final do trabalho percebendo que a sobrecarga nas operações não é significativa e que os algoritmos da GR-Tree funcionam melhor que os da GR-Tree intermediária.

Os algoritmos para Inserção, Remoção e Busca são os mesmos da R*-Tree com alterações feitas nas políticas de armazenamento e divisão dos nós. Os algoritmos alterados são apenas o *ChooseSubtree*, *Split* e *RemoveTop* para que possam levar em conta essas modificações que foram feitas nos *MBR*s tornando-os regiões envolventes não retangulares.

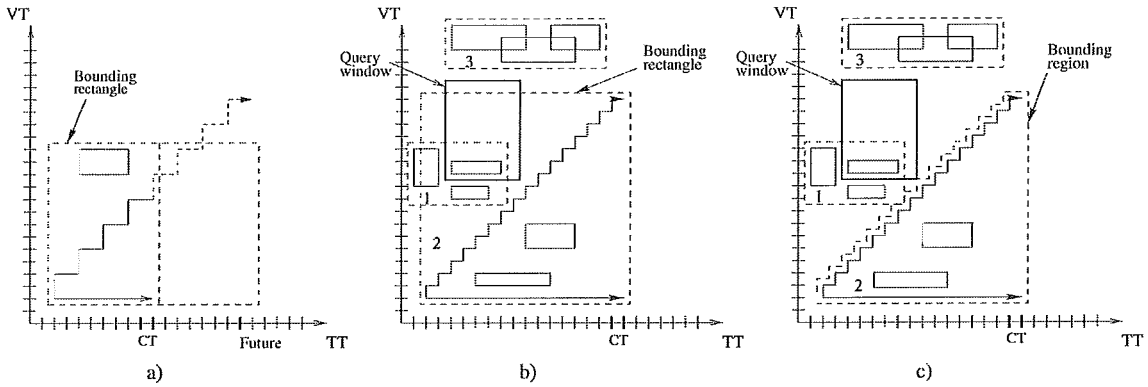


Figura 12. ***Interseção entre dois MBRs (a) Antes da inserção da nova entrada (b) Depois da inserção da nova entrada no nó 1. (c) Depois da inserção da nova entrada no nó 2. (d) Caso (b) após um período de tempo. (e) Caso (c) após um período de tempo.***

Como exemplo, analise o caso da Figura 12 retirado diretamente de BLIUJUTE et al. (1998). A parametrização por menor acréscimo de área de sobreposição para se inserir um objeto em um nó não é suficiente tendo em vista que o tempo "agora" é crescente. Pode-se perceber nesta figura que após um período de tempo a escolha feita pelo algoritmo *ChooseSubtree* da R*-Tree teria feito uma má escolha.

Sob o ponto de vista dos algoritmos de Inserção, Remoção e Busca, temos quatro tipos de região bitemporal:

1. Retângulos estáticos e regiões em formato de escada estáticas.

2. Retângulos crescendo na direção do tempo de transação (com *tt_morte=UC*)

3. Regiões em formato de escada ainda em crescimento (com *tt_morte=UC* e *vt_morte=NOW* e o indicador *Rectangle* ativado)

4. Retângulos ainda em crescimento nas duas direções (com *tt_morte=UC* e *vt_morte=NOW* e o indicador *Rectangle* desativado)

Estes tipos de região bitemporal encontram-se em ordem de prioridade, ou seja, é preferível se encontrar nós do tipo 1 a encontrar nós do tipo 2 e assim por diante. Os algoritmos *ChooseSubtree*, *Split* e *RemoveTop* da R*-Tree devem ser modificados para tentar priorizar os tipos de nós resultantes.

O algoritmo *ChooseSubtree* modificado tenta então priorizar inicialmente o tipo de nó em que uma entrada deve ser inserida. Depois, de posse de um grupo de nós do mesmo tipo, aplica-se a heurística do *ChooseSubtree* original da R*-Tree (que prioriza o aumento de área de sobreposição ao se inserir o objeto) para se decidir em qual nó será inserido o objeto. Este algoritmo modificado desce na árvore selecionando os nós do mesmo tipo onde melhor pode ser inserido o objeto, ou seja, os nós do mesmo tipo que, quando inserido o objeto, mantenham o tipo. Se vários grupos de nós de diferentes tipos forem escolhidos, prioriza-se o grupo com menor tipo. Se nenhum grupo de nós for encontrado, o algoritmo escolhe o grupo de nós do mesmo tipo que, quando inserido o objeto, terá seu tipo aumentado ao mínimo. Se existirem vários grupos de tipos diferentes, o algoritmo escolhe o grupo de pior tipo.

O algoritmo de divisão de nós (*Split*) modificado tenta, da mesma forma que o algoritmo *ChooseSubtree*, dividir o nó em dois nós com os melhores tipos possíveis. Para um melhor entendimento do algoritmo de divisão de nós modificado ver o algoritmo no artigo da GR-Tree em BLIUJUTE et al. (1998).

O algoritmo *RemoveTop* modificado tenta inicialmente remover as *p%* piores entradas encontradas no nó relacionando a pior entrada ao tipo. Depois, um algoritmo de complexidade quadrática é utilizado para remover entradas de forma que minimize a área da região envolvente do nó.

3.3. Métodos de Acesso Espaçotemporais

Aqui nesta seção, seguindo os passos da seção anterior, iremos dividir os métodos de acesso segundo o tipo de tempo que eles suportam: tempo de transação, tempo de validade ou ambos.

3.3.1. Métodos de Acesso Bitemporais

3.3.1.1. R^{ST}-Tree

Este método de acesso proposto em SALTENIS et al. (2000) permite o armazenamento eficiente de dados espaço-bitemporais. É uma extensão da GR-Tree para dados espaciais. Os algoritmos *ChooseSubtree*, *Split* e *RemoveTop* originais da R*-Tree também foram alterados de forma um pouco diferente da GR-Tree para que espacialidade dos dados pudesse também interferir na escolha da inserção de entradas e da divisão dos nós.

3.3.2. Métodos de Acesso com Suporte a Tempo de Transação

3.3.2.1. Overlapping Linear Quadtree (OLQT)

A OLQT (TZOURAMANIS et al., 1998) é uma sobreposição de Quadtrees (SAMET, 1990) baseada nos mesmos princípios da OBT. As quadtrees são estruturas de dados utilizadas largamente para o armazenamento de imagens raster e essas estruturas fogem do escopo deste trabalho e não serão portanto discutidas detalhadamente.

3.3.2.2. Multiversion Linear Quadtree (MLQT)

A Linear Quadtree (SAMET, 1990) é uma estrutura de dados especial criada para utilização dos dados em memória principal. Seu armazenamento em memória secundária é feito utilizando-se uma B+-Tree onde o valor de cada nó preto da Quadtree é armazenado e uma chave de mapa de bits é criada segundo sua posição corresponde na Quadtree. Isto pode ser visto na Figura 13 abaixo.

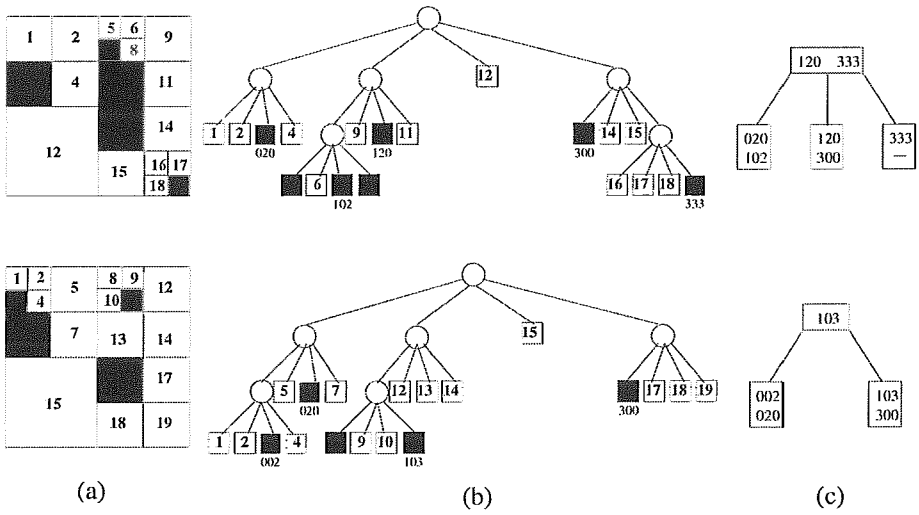


Figura 13. ***Duas imagens similares de 2^3x2^3 pontos (a) e suas correspondents Quadtrees de Região (b) e Quadtrees Lineares de Região (c).***

O que foi proposto em (TZOURAMANIS et al., 2000) é utilizar-se de uma MBVT para armazenar Linear Quadtrees ao invés de B+-Trees, e assim transformar diretamente uma estrutura espacial em uma estrutura espaçotemporal parcialmente persistente com tempo de transação armazenado.

3.3.2.3. MR-Tree

A MR-Tree (XU et al., 1990) é uma estrutura que funciona exatamente da mesma forma que a OBT e a OLQT no contexto de dados espaciais utilizando-se R-Trees, ou seja, é uma árvore de sobreposição baseada na R-Tree. As modificações efetuadas nos algoritmos da R-Tree seguem os mesmos princípios de árvores sobrepostas, onde para cada nova versão da estrutura cria-se um novo nó raiz (e armazena-se em uma lista de nós raízes) duplicando-se os nós apenas onde houve alterações. Desta forma economiza-se bastante espaço comparando-se com a idéia acima de uma R-Tree para cada versão do conjunto de dados, mas o desperdício de espaço ainda é considerável, uma vez que todo o caminho desde o nó raiz até a folha em uma modificação é duplicado gerando uma grande quantidade de duplicação de nós e ainda, esta característica tende a degradar ainda mais o desempenho das consultas conforme o número de alterações na base de dados e também conforme o número de versões existentes.

3.3.2.4. RT-Tree

A RT-Tree proposta em XU et al. (1990) pelos mesmos autores da MR-Tree é uma tentativa de tornar o desperdício de espaço com duplicação de nós menor e ainda melhorar o desempenho no processamento de consultas de uma forma geral, visto que para consultas por instante de tempo a MR-Tree é muito mais eficiente.

As informações de tempo de vida de um objeto são armazenadas nas entradas dos nós junto com a informação espacial de forma que exista somente uma árvore na estrutura de dados. Uma entrada na RT-Tree toma então a forma < *MBR*, *ponteiro, t_nasc, t_morte* > e a árvore possui uma altura um pouco maior do que a R-Tree correspondente para a versão corrente. Devido à esse fato o desempenho das consultas por instante de tempo ficam um pouco prejudicadas comparando-se com a MR-Tree.

São propostas três formas de se fazer uma divisão de nós: divisão por espaço, divisão por tempo e divisão por características semânticas dos dados. No primeiro caso, a divisão dos nós é feita exatamente como o é na R-Tree; no segundo, uma divisão pelo intervalo de tempo do nó é executada e esta divisão pode facilitar consultas envolvendo intervalos de tempo, pois objetos com tempos semelhantes são armazenados juntos nos mesmos nós; já no terceiro caso a divisão é baseada em alguma semântica previamente conhecida sobre o conjunto de dados e é muito difícil de ser implementado pois a estrutura perde sua generalidade. Desta forma, como foi proposta a estrutura, ou ela é organizada espacialmente ou temporalmente e não existe uma forma de se aliar essas duas características no algoritmo de divisão de nós e conseqüentemente no algoritmo de consultas. Consultas temporais em RT-Trees organizadas espacialmente bem como consultas espaciais em RT-Trees organizadas temporalmente sofrem diretamente com esse problema.

3.3.2.5. 2+3D R-Tree

A estrutura com duas R-Trees apresentada anteriormente em KUMAR et al. (1995) e em KUMAR et al. (1997) para bancos de dados bitemporais pode funcionar perfeitamente para bancos de dados espaçotemporais com suporte ao tempo de transação. A primeira R-Tree armazena objetos bidimensionais com *t_morte* indeterminado. A segunda R-Tree armazena os objetos bidimensionais com *t_morte* definido e mais um eixo é incluído para o armazenamento do tempo de transação dos

objetos. Logo temos duas R-Trees, uma bidimensional e outra tri-dimensional, de onde vem o nome 2+3 R-Tree.

Esta estrutura não possui o problema de tempo de validade "agora" já que não armazena o tempo de validade, e mantém o problema de uma busca ter que ser percorrida em duas árvores.

3.3.2.6. HR-Tree

A HR-Tree foi proposta em NASCIMENTO et al. (1998) e representa uma evolução da MR-Tree para a utilização de Hilbert R-Trees. Funciona exatamente como a MR-Tree onde novos nós raízes vão sendo criados incrementalmente com alterações em novas versões da estrutura e apenas os braços da árvore que foram modificados são duplicados, sendo mantidos intactos aqueles que não sofreram modificação. Cabe aqui ressaltar que como a Hilbert R-Tree utiliza o(s) nó(s) vizinho(s) quando ocorre um overflow, estes nós também têm que ser duplicados. A Figura 14 retirada de NASCIMENTO et al. (1998) mostra um exemplo com duas alterações feitas em nós de uma HR-Tree: no tempo T1, é feita uma alteração no objeto 3 gerando o objeto3a, no tempo T2, o objeto 8 dá lugar ao objeto 8a.

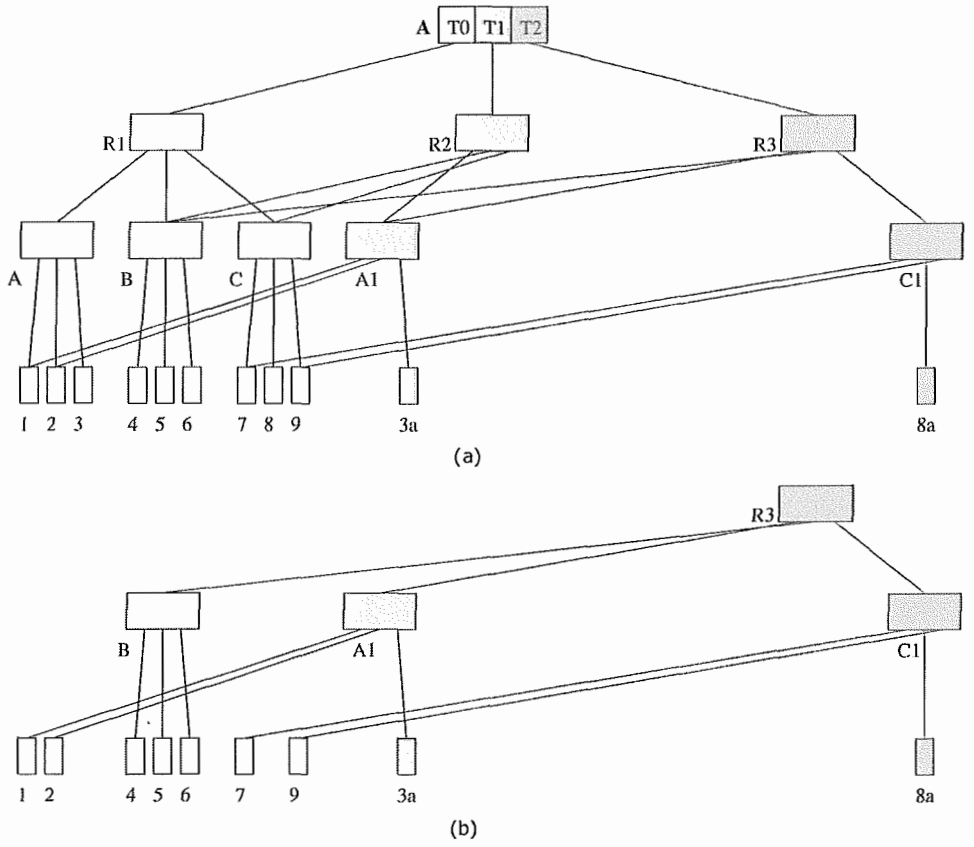


Figura 14. ***Exemplo da estrutura de uma HR-Tree. (a)Visão física da estrutura. (b) Visão lógica da estrutura.***

3.3.2.7. PPR-Tree

A PPR-Tree (do inglês Partially Persistent R-Tree) foi proposta em KOLLIOS et al. (2001) e é uma extensão dos métodos da Bitemporal R-Tree para suportar dados espaciais. Da forma como foi proposta, é uma estrutura para o armazenamento de tempo de transação e não mais bitemporal. É mostrado neste trabalho que a política de *merge* de nós que minimiza o perímetro (política iii) é a que dá melhores resultados.

3.4. Sumário

Neste capítulo fizemos uma vasta revisão bibliográfica dos métodos de acesso espaciais, temporais e espaçotemporais pertinentes ao desdobramento desta dissertação. Procuramos mostrar as principais características de cada método de acesso ressaltando suas qualidades e defeitos quando necessário.

No próximo capítulo será discutida a proposta desta dissertação: a TR-Tree. Serão apresentados e discutidos sua estrutura e seus principais algoritmos.

4. Temporal R-Tree (TR-Tree)

Neste Capítulo iremos descrever com maiores detalhes a estrutura de índice proposta nesta dissertação, a R-Tree Temporal ou TR-Tree (do inglês *Temporal R-Tree*). A TR-Tree é um método de acesso espaçotemporal com suporte a tempo de transação. A estrutura e os principais algoritmos propostos para a TR-Tree encontram-se em ZIMBRÃO et al. (1999). A Seção 4.1 mostrará com detalhes a estrutura da TR-Tree. A Seção 4.2 irá listar os algoritmos básicos de inserção, remoção e de consultas de seleção. A Seção 4.3 irá falar sobre o problema da junção espaçotemporal e irá mostrar um algoritmo que resolve este problema na TR-Tree.

4.1. Estrutura da TR-Tree

A TR-Tree é uma especialização da MVBT para o tratamento de dados espaciais, ou seja, alteramos a estrutura e os algoritmos da MVBT para que a mesma possa utilizar uma estrutura de dados espacial, a R-Tree, mais especificamente a R*-Tree. Conseqüentemente, a TR-Tree é uma versão espacial da MVBT capaz de armazenar e consultar eficientemente dados espaçotemporais para tempo de transação.

A TR-Tree, assim como a MVBT, é então um grafo acíclico e direcionado. É um grafo pois não possui apenas um nó raiz como todas as árvores, mas sim uma seqüência deles com seus respectivos tempos de vida formando uma partição do tempo total da estrutura. Estes nós encontram-se armazenados em uma estrutura a parte e estão indexados pelos seus respectivos tempos de vida. Chamaremos esta estrutura de vetor de nós raízes (*root index structure*).

A estrutura dos nós da TR-Tree é bastante similar à da R-Tree com algumas alterações para que possa tratar de informações temporais. Devido ao fato de um nó poder conter entradas em diferentes versões, é necessário que se armazene nos mesmos o tempo de vida e as entradas ficam então na forma *<MBR, ponteiro, t_nasc, t_morte>*. Além da informação temporal armazenada nas entradas, há a necessidade do armazenamento da informação de seu tempo de nascimento nos nós para que as alterações em bloco se tornem possíveis. A permissão de operações em bloco na TR-Tree é uma importante melhoria em relação a MVBT, onde cada operação gera uma

nova versão dos dados. Isto não é condizente com o formato das operações em uma transação onde executa-se normalmente mais de uma operação na mesma transação.

Tabela 3. Tipos de nós e entradas

Tipo de Nó (Entrada)	Descrição	Característica
Nó (entrada) morto(a)	Nó (entrada) que possui tempo de morte determinado(a), que já foi removido(a) da base de dados ou foi duplicado(a).	*t_morte* ≠ '*'
Nó (entrada) vivo(a)	Nó (entrada) que possui tempo de morte indeterminado(a).	*t_morte* = '*'
Nó (entrada) novo(a)	Nó (entrada) vivo(a) que nasceu no tempo corrente da estrutura.	*t_morte*='*' **and** *t_nasc* = *t_atual*
Nó (entrada) antigo(a)	Nó (entrada) vivo(a) que nasceu em um tempo anterior ao tempo atual da estrutura, ou seja,.	*t_morte*='*' **and** *t_nasc* < *t_atual*

A eficiência de tempo e espaço da R-Tree é baseada nas seis (**R1...R6**) propriedades propostas por GUTTMAN (1984). Para manter essa eficiência, as propriedades da R-Tree devem ser generalizadas para tratar da existência de entradas em diferentes versões em um mesmo nó da árvore. Seja M o número máximo de entradas em um nó e $m = M / k$ um parâmetro especificando o número mínimo de entradas vivas em um nó, onde k é uma constante, $k \geq 2$. Veja na Tabela 3 uma discussão sobre nós(entradas) mortos(as) e vivos(as). As seis propriedades da R-Tree devem ser modificadas (em negrito) como se segue:

R1. Cada nó folha contém no mínimo m entradas **vivas** e no máximo M entradas ao menos que seja um nó raiz;

R2. Para cada registro de índice < *MBR, ponteiro_objeto, t_nasc, t_morte* > em um nó folha, MBR é o menor retângulo que espacialmente contém os objetos de dados n-dimensionais representados pela tupla indicada;

R3. Cada nó interno contém no mínimo m entradas **vivas** para nós filhos e no máximo M ao menos que seja um nó raiz;

R4. Para cada entrada < *MBR, ponteiro_filho, t_nasc, t_morte* > em um nó interno, MBR é o menor retângulo que espacialmente contém todos os retângulos do nó filho **no intervalo de tempo [t_nasc, t_morte)**;

R5. O nó raiz tem pelo menos dois filhos **vivos** a menos que seja uma folha;

R6. Todas as folhas **de uma mesma árvore** aparecem no mesmo nível.

Note que as propriedades **R1** e **R3** são praticamente as mesmas (também o são **R2** e **R4**) diferindo apenas se estamos nos referindo a nós folha ou nós internos e conseqüentemente se os ponteiros nas entradas referem-se a ponteiros para a posição real do objeto espacial (nós folha) ou para o nó filho na árvore (nós internos). Iremos tratar, sem perda de generalidade, destes dois tipos de nós da mesma forma, ou seja, quando tratarmos de registros de índices em nós folha, também os chamaremos de entradas e o *ponteiro_objeto* e *ponteiro_filho* serão somente chamados de *ponteiro*.

Assim como na MVBT, chamamos o conjunto das propriedades **R1**, **R3** e **R5** de condição de versão fraca (*weak version condition*). Note que uma entrada viva significa que ela está viva durante o tempo de vida do nó. Também similarmente à MVBT, iremos definir uma nova propriedade **R7** que deve ser satisfeita após uma mudança estrutural, e iremos chamá-la de condição de versão forte (*strong version condition*).

R7. O número de entradas em um nó após uma mudança estrutural deve estar entre $(1+\varepsilon)\times m$ e $(k-\varepsilon)\times m$, onde ε é uma constante de ajuste, $\varepsilon>0$.

Quando as propriedades R1 e R3 são violadas uma mudança estrutural na árvore ocorre e, antes de executar as operações de divisão de nós (*overflow*) ou de reinserção de entradas (*underflow*), uma operação chamada divisão de versão é efetuada. Esta operação cria um novo nó e copia todas as entradas vivas do nó anterior para este novo nó criado. Esta operação pode ser vista no exemplo mostrado na Figura 7. É a operação de divisão de versão que gera duplicação de entradas na estrutura e precisa ser evitada ao máximo. Com os testes de condição de versão forte feitos após as mudanças estruturais garantimos que, após uma mudança estrutural em um nó, pelo menos $\varepsilon\times m+1$ inserções ou remoções de entradas ainda podem ser executadas naquele nó antes que a próxima mudança estrutural se torne necessária. A escolha de ε então pode funcionar como um parâmetro de ajuste de desempenho da estrutura. Em nossos testes, utilizamos

α=0.5, k=3 e ε=0.3. Assim, para a capacidade de um nó de 90 entradas, os parâmetros da TR-Tree ficam como mostrado na Tabela 4 e na Figura 15.

Tabela 4. *Parâmetros da TR-Tree*

Condição fraca de versão	Max. entradas/nó	90
	Min. entradas vivas/nó	30
Condição forte de versão	Max. entradas/nó	81
	Min. entradas vivas/nó	39

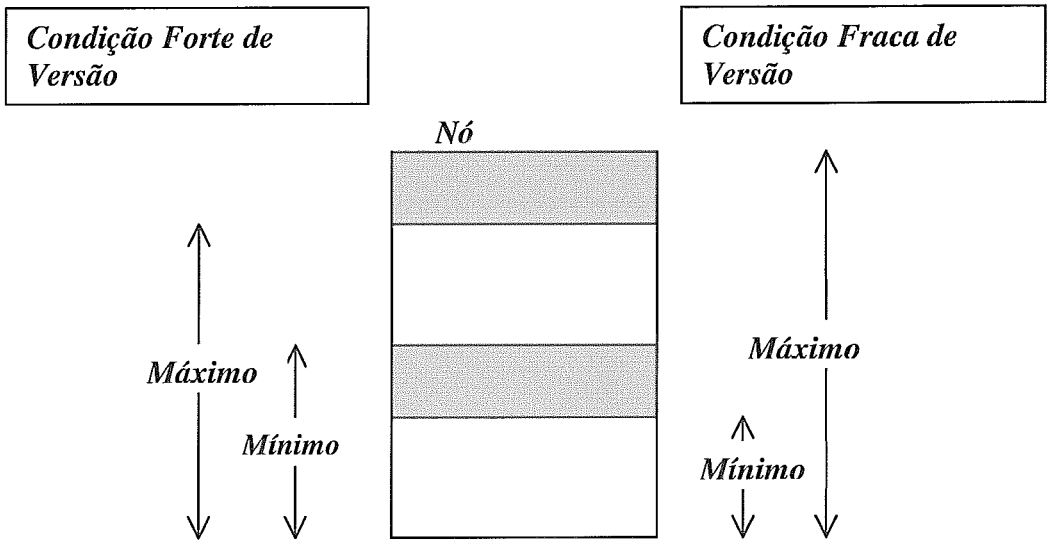


Figura 15. *Número de entradas por nó para condições de versão fraca e forte*

Desta forma, neste exemplo citado, após uma divisão de versão (que pode ou não ser seguida de uma divisão de nó) podem ser efetuados pelo menos mais 9 alterações naquele nó antes que ocorra um overflow ou underflow fraco de versão. Isto garante que a maioria das alterações na TR-Tree não irão disparar novas mudanças estruturais, o que é indesejado, pois gera muita duplicação de dados.

Uma análise teórica de eficiência das R-Trees, feita em BECKER et al. (1996), mostra que os algoritmos de busca por instante de tempo, por intervalo de tempo e de alterações (inserções e remoções) são assintoticamente ótimos. Esta mesma análise não pode ser feita para a TR-Tree, pois algumas premissas utilizadas nesta análise não podem ser estendidas para o contexto de dados espaciais utilizando-se R-Trees. Espera-se, entretanto, que a TR-Tree possua as mesmas boas propriedades para os algoritmos de alterações e buscas da MVBT.

4.2. Algoritmos de Busca, Remoção e Inserção

Nesta seção iremos apresentar os algoritmos da TR-Tree para inserção, remoção e busca por instante de tempo e por intervalo de tempo. Note que não há algoritmo para modificações, uma vez que uma modificação em um objeto gera uma remoção (lógica) seguida de uma inserção. Essas operações são correspondentes na TR-Tree às operações básicas da R-Tree.

4.2.1. Inserção

A inserção na TR-Tree recebe como argumentos um MBR e um identificador do objeto que pode ser um ponteiro físico/lógico onde o objeto espacial pode ser encontrado. Inserções podem ser feitas apenas nos nós folha da R-Tree corrente da TR-Tree, ou seja, como uma estrutura de dados parcialmente persistente, alterações são feitas sempre na versão corrente da mesma. Após uma inserção, o tempo corrente da TR-Tree pode ser incrementado ou não, permitindo assim operações em bloco no mesmo instante de tempo, no caso de não incremento do tempo atual. A MVBT não dá suporte a operações em bloco no mesmo instante de tempo e alguns mecanismos adicionais tiveram que ser implementados na TR-Tree para tal melhoria.

Inicialmente, uma busca é executada para encontrar o nó folha ótimo onde a entrada para o MBR dado será inserida. Esta busca é feita apenas no último nó raiz da TR-Tree, e deve considerar apenas entradas e nós vivos. Esta busca utiliza o mesmo algoritmo *ChooseSubtree* da R*-Tree (BECKMANN et al., 1990).

A nova entrada é então inserida no nó escolhido no passo anterior. Se houver espaço para a entrada, o algoritmo necessita subir até o nó raiz da árvore ajustando os MBRs caso necessário. Caso não haja espaço, devemos proceder de forma diferente caso o nó seja novo ou antigo. Veja na Tabela 3 a diferença existente entre um nó novo e um nó antigo.

- Se o nó for novo, ou seja, todas as entradas neste nó foram criadas necessariamente no tempo atual e não há a necessidade de uma divisão de versão. Será efetuado então uma divisão de nó ou uma reinserção forçada, como o faz o algoritmo de inserção da R*-Tree. Uma reinserção forçada ocorre apenas uma vez em cada nível

da árvore durante o algoritmo de inserção. Pode-se ver que ao início do algoritmo de inserção, de posse do nó raiz atual, limpa-se os flags de reinserção forçada. Neste caso, se ainda não tiver ocorrido uma reinserção forçada neste nível da árvore, remove-se $p\%$ das entradas do nó em overflow colocando-as na lista de reinserção, marca-se o flag de reinserção forçada para este nível e sobe-se até o nó raiz da árvore ajustando os MBRs quando necessário. Se não for efetuada uma reinserção forçada de entradas, uma divisão de nó é efetuada. É importante notar que nunca ocorre este caso na inserção de uma nova entrada, mas sim nas reinserções e na recursividade do algoritmo, pois ao início do algoritmo todos os flags de reinserção são limpados. Utilizamos o algoritmo *AxisSplit* da R*-Tree (BECKMANN et al., 1990) para efetuar a divisão de nó. Neste algoritmo um nó se divide em dois novos nós, logo temos que, no nó pai do nó em overflow, remover a entrada para o nó antigo e inserir duas entradas para os dois novos nós criados. Esta operação possui o mesmo significado que apenas inserir uma nova entrada para um novo nó e aproveitar a entrada antiga para o outro nó gerado pelo algoritmo de divisão de nó. Esta inserção no nó pai é efetuada de forma recursiva.

- Se o nó for antigo, ou seja, tenha sido criado em um tempo menor que o tempo atual da árvore, ele pode conter entradas mortas. Uma divisão de versão é necessária e um novo nó é criado contendo apenas as entradas vivas do nó em overflow. Após a divisão de versão, três casos podem ocorrer:

  Caso 1. O nó não violar nenhuma condição (forte). Neste caso deve-se, no nó pai do nó em overflow, atualizar o MBR deste nó em overflow e inserir uma entrada para o novo nó criado após a divisão de versão.

  Caso 2. O nó violar o limite superior da condição forte de versão (dizemos que encontra-se em overflow de condição forte de versão). Devemos, neste caso, efetuar ou uma divisão de nó ou uma reinserção forçada de entradas de forma semelhante ao caso em que o nó em overflow é novo. A diferença fica por conta de ter sido previamente efetuado uma divisão de versão, logo o nó em overflow não será fisicamente removido da árvore. Desta forma, o algoritmo de reinserção forçada irá, no pai do nó em overflow, inserir uma nova entrada (ao contrário do caso anterior onde ele apenas ajustava o MBR do nó em overflow) e o algoritmo

de divisão de nó irá inserir duas novas entradas (ao contrário do caso anterior que inseria apenas uma, uma vez que o nó em overflow estava sendo removido).

Caso 3. O nó violar o limite inferior da condição forte de versão (dizemos que encontra-se em underflow de condição fraca de versão). Neste caso, devemos inserir as entradas deste novo nó em underflow na lista de reinserção e atualizar os MBRs até a raiz caso necessário. Este novo nó criado pela divisão de versão não será, portanto, acoplado à árvore.

Ao final destes passos, devemos processar a lista de reinserção. Ao processar a lista de reinserção, não devemos nos preocupar com o número mínimo de entradas no nó raiz pois este número pode vir a ser restaurado até o final da lista. Só olhamos este valor ao final de todo o processamento da lista de reinserção e assim possivelmente diminuímos a altura da árvore.

Em seguida encontram-se os algoritmos pertinentes ao processo de inserção descrito acima.

```
Procedimento Inserir( E: Entrada )
Inicio
  Encontrar o nó raiz R na versão atual da TR-Tree
  Limpar os flags de reinserção forçada
  N = ChooseSubtree(R) // R*-Tree ChooseSubtree
  InserirEntrada(N, E)
  TratarAlteracao_MBR_AteRaiz( N, L )
  Se not Vazia(L) Então
    ProcessaListaReinsercao(L)
  Se Underflow(R) Então
    TratarUnderflowRaiz(R)
Fim
```

Algoritmo 1. Algoritmo de Inserção – Procedimento Inserir

```
Procedimento TratarAlteracao_MBR_AteRaiz( N: Nó; L: ListaReinsercao )
Inicio
  Enquanto Nível(N) > 0 Faça
    Se Overflow(N) Então
      TratarOverflow(N, A, B, L)
      N = SubirNivel(MATAR_ENTRADA, N, A, B)
    Senão Se Underflow(N) Então
      TratarUnderflow(N, L)
      N = SubirNivel(MATAR_ENTRADA, N, Ø, Ø)
    Senão
      N = SubirNivel(ALTERAR_MBR, Ø, Ø, Ø)

  Se Overflow(N) Então
    TratarOverflowRaiz(N)
Fim
```

Algoritmo 2. Algoritmo de Inserção – Procedimento TratarAlteracao_MBR_AteRaiz

```
Procedimento TratarOverflow( N, A, B: Nó; L: ListaReinsercao )
Inicio
  Se Novo(N) Então
    Se not FlagsReinsercao[Nível(N)] Então
      ReinsercaoForcada(N, L)
    Senão
      DivisaoNo(N, A, B)
  Senão
    DivisaoVersao(N, X)
    Se OverflowCondicaoForteVersao(X) Então
      Se not FlagsReinsercao[Nível(N)] Então
        ReinsercaoForcada(X, L)
        A = X
      Senão
        DivisaoNo(X, A, B)
    Senão Se UnderflowCondicaoForteVersao(X) Então
      RemoverEntradas(X, L)
    Senão
      A = X
Fim
```

Algoritmo 3. Algoritmo de Inserção – Procedimento TratarOverflow

```
Procedimento TratarOverflowRaiz( N: Nó )
Inicio
  Se Novo(N) Então
    DivisaoNo(N, A, B)
    InserirEntrada(N, CriarEntrada(A))
    InserirEntrada(N, CriarEntrada(B))
  Senão
    DivisaoVersao(N, X)
    Se OverflowCondicaoForteVersao(X) Então
      DivisaoNo(X, A, B)
      InserirEntrada(X, CriarEntrada(A))
      InserirEntrada(X, CriarEntrada(B))
      CriarNovoRoot(X)
Fim
```

Algoritmo 4. Algoritmo de Inserção – Procedimento TratarOverflowRaiz

```
Procedimento TratarUnderflow( N: Nó; L: ListaReinsercao )
Inicio
  Se Novo(N) Então
    RemoverEntradas(N, L)
  Senão
    DivisaoVersao(N, X)
    RemoverEntradas(X, L)
Fim
```

Algoritmo 5. Algoritmo de Inserção – Procedimento TratarUnderflow

```
Procedimento TratarUnderflowRaiz( N: Nó )
Inicio
  Se Novo(N) Então
    Seja F o único nó filho de N
    Se Novo(F) Então
      F passa a ser o nó raiz da árvore corrente no lugar de N
    Senão
      DivisaoVersao(F, X)
      X passa a ser o nó raiz da árvore corrente no lugar de N
  Senão
    DivisaoVersao(N, X)
    Seja F o único nó filho de X
    Se Novo(F) Então
      F passa a ser o nó raiz da árvore corrente no lugar de N
    Senão
      DivisaoVersao(F, Y)
      Y passa a ser o nó raiz da árvore corrente no lugar de N
      MatarEntrada(N, F)
Fim
```

Algoritmo 6. Algoritmo de Inserção – Procedimento TratarUnderflowRaiz

```
Função SubirNivel( op: OperacaoSubirNivel; N, A, B: Nó ) : Nó
Inicio
  Se op = ALTERAR_MBR Então
    P = Pai(N)
    MBR(P) = MBR(P) U MBR(N)
  Senão Se op = MATAR_ENTRADA Então
    P = Pai(N)
    Seja E a entrada para N em P
    MatarEntrada(P, E)
    Se A ≠ ∅ Então
      InserirEntrada(P, CriarEntrada(A))
    Se B ≠ ∅ Então
      InserirEntrada(P, CriarEntrada(B))
  Retornar P
Fim
```

Algoritmo 7. Algoritmo de Inserção – Função SubirNivel

```
Procedimento ProcessaListaReinsercao( L: ListaReinsercao )
Inicio
  Enquanto not Vazia(L) Faça
    E = Remover(L)
    Reinserir(E)
Fim
```

Algoritmo 8. Algoritmo de Inserção – Procedimento ProcessaListaReinsercao

Alguns comentários sobre os algoritmos ainda devem ser tecidos:

- O procedimento *ChooseSubtree* é um algoritmo similar ao algoritmo *ChooseSubtree* da R*-Tree detalhado na Seção 3.1.2 e por isso foi omitido.

- Os procedimentos *ReinsercaoForcada* e *DivisaoNo* fazem exatamente o que fazem os mesmos na R*-Tree e por isso foram omitidos. O procedimento *ReinsercaoForcada* remove as entradas no nó e as insere na lista de reinserção.

Este procedimento ainda marca o flag de reinserção para que não aconteçam mais reinserções forçadas naquele nível da árvore dentro da mesma inserção O procedimento *DivisaoNo* é equivalente ao algoritmo *AxisSplit* detalhado na Seção 3.1.2.

- O procedimento *Reinserir* chamado no procedimento *ProcessaListaReinsercao* foi também omitido por ser muito semelhante ao procedimento Inserir. A diferença fica por conta da altura das entradas: no procedimento Inserir as entradas são sempre inseridas nos nós folha da árvore corrente e no procedimento *Reinserir* as entradas devem ser reinseridas na mesma altura em que foram removidas da árvore (repare que esta altura é contabilizada de baixo para cima).

- O procedimento *DivisaoVersao* foi omitido devido à sua simplicidade e ao fato de já termos citado o mesmo diversas vezes anteriormente nesta dissertação. Este procedimento copia para um nó novo e vazio todas as entradas vivas de um determinado nó.

- Os procedimentos *InserirEntrada*, *MatarEntrada*, *RemoverEntradas*, *CriarEntrada* foram também omitidos devido às suas simplicidades. Imaginamos que a semântica de nomenclatura com seus parâmetros identifiquem exatamente o que estes procedimentos executam. O procedimento *InserirEntrada* recebe como parâmetros um nó N e uma entrada E e apenas insere E no nó N. O procedimento *MatarEntrada* recebe como parâmetros um nó N e uma entrada E e coloca em E.t_morte o tempo atual da estrutura. O procedimento *RemoverEntradas* recebe como parâmetros um nó N e a lista de reinserção L e remove todas as entradas em N inserindo-as em L. A função *CriarEntrada* recebe como parâmetro um nó N e retorna uma entrada com o MBR contendo todos os MBRs das entradas de N e um ponteiro para N, com t_nasc igual ao tempo corrente da estrutura e t_morte = '*'.

4.2.2. Remoção

A remoção na TR-Tree recebe como argumentos um MBR e um identificador do objeto espacial a ser removido. Inicialmente uma busca é feita para encontrar o nó que

contém uma entrada para estes MBR e identificador de tupla. Se este nó não for encontrado, a remoção é abortada. Como o algoritmo de inserção, esta busca é feita apenas no último nó raiz da TR-Tree, pois alterações são sempre feitas na versão atual da estrutura.

Caso este nó encontrado seja novo, a entrada é fisicamente removida do mesmo, uma vez que a TR-Tree não armazena estados intermediários. Caso contrário ela é apenas marcada como morta no tempo atual. Se o nó não se encontra em underflow de condição fraca de versão, apenas sobe-se até a raiz da árvore ajustando os MBRs quando necessário. Caso contrário, novamente temos dois possíveis tratamentos:

- O nó em underflow é novo. Todas as entradas deste nó foram necessariamente criadas no tempo atual, logo devemos inserir todas na lista de reinserção. Subir na árvore ajustando os MBRs quando necessário.

- O nó em underflow é antigo. É necessária uma divisão de versão, pois o nó pode conter entradas mortas. Após a divisão de versão, o nó necessariamente encontra-se em underflow de condição forte de versão, uma vez que anteriormente já se encontrava em underflow de condição fraca de versão. Todas as entradas do novo nó criado pela divisão de versão são inseridas na lista de reinserção. Este novo nó criado não será, portanto acoplado à árvore. Deve-se marcar como morta a entrada no nó pai para o nó em underflow utilizando recursivamente este mesmo algoritmo de remoção de entradas.

Ao final destes passos processa-se a lista de reinserção da mesma forma como no algoritmo de inserção. Segue abaixo o algoritmo de remoção de um objeto na TR-Tree. Repare que os outros algoritmos pertinentes ao processo de remoção de objetos já foram mostrados e comentados anteriormente no algoritmo de inserção.

```
Procedimento Remover(E: Entrada)
Inicio
  Encontrar o nó raiz R na versão atual da TR-Tree
  Limpar os flags de reinserção forçada
  N = EncontrarEntrada(R, E)
  Se N ≠ Ø Então
    MatarEntrada(N, E)
  TratarAlteracao_MBR_AteRaiz(N, L)
  Se not Vazia(L) Então
    ProcessaListaReinsercao(L)
  Se Underflow(R) Então
    TratarUnderflowRaiz(R)
Fim
```

Algoritmo 9. Algoritmo de Remoção – Procedimento Remover

As mesmas considerações feitas sobre os procedimentos e funções omitidas nas explicações sobre o algoritmo de inserção também valem para o algoritmo de remoção.

4.2.3. Busca por instante de tempo

O algoritmo de busca em instante de tempo recebe como argumentos um retângulo de busca (janela) *R* e um tempo de busca *t*. É basicamente o mesmo algoritmo de busca da R*-Tree, mas com um passo inicial que busca o nó raiz apropriado, isto é, o nó raiz cujo intervalo de tempo de vida contém o tempo t. Como o vetor de nós raiz é uma estrutura de memória ordenada por *t_nasc* e não existem intervalos com interseção, uma busca binária pode aqui ser realizada. Uma vez encontrado o nó raiz a busca prossegue da mesma forma que na R*-Tree, apenas ignorando as entradas em que o intervalo de tempo de vida não contém t.

Segue abaixo o algoritmo que executa a busca por instante de tempo.

```
Procedimento BuscaInstante( R: MBR; t: Tempo )
Início
  Encontrar o nó raiz N correspondente ao tempo t no vetor de nós raízes
  BuscaInstanteNoRaiz( N, R, t )
Fim

Procedimento BuscaInstanteNoRaiz( N: Nó; R: MBR; t: Tempo )
Início
  Para cada entrada E em N Faça
    Se E.MBR ∩ R E t ∈ [E.t_nasc, E.t_morte) Então
      Se N é folha Então
        Retornar E como resultado
      Senão
        BuscaInstanteNoRaiz( N.ponteiro, R, t )
Fim
```

Algoritmo 10. Algoritmo de Busca por Instante de Tempo – Procedimento BuscaInstante

4.2.4. Busca por intervalo de tempo

A consulta por intervalo de tempo recebe como parâmetros um retângulo R e um intervalo de tempo de busca $[t_1, t_2)$. Devido às divisões de versão que duplicam entradas, esta consulta necessita de uma estratégia especial para que não haja acessos a disco desnecessários e/ou repetições de resultados.

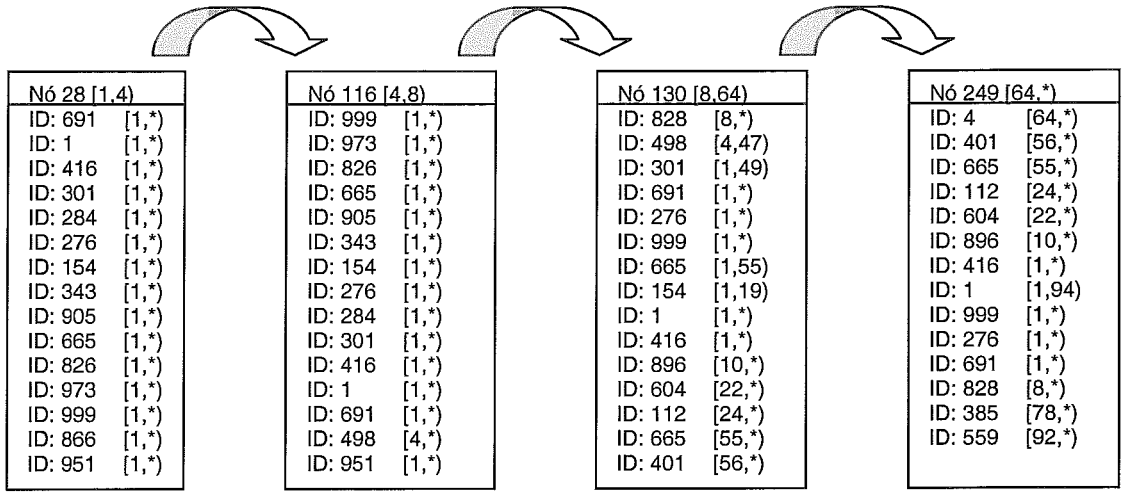
Uma estratégia ingênua de se fazer este tipo de consulta seria a de pegar todos os nós raízes cujos intervalos de tempo de vida incluem $[t_1, t_2)$ e seguir executando as buscas por cada árvore independentemente. Esta busca é ineficiente porque estaremos percorrendo entradas mais de uma vez e os resultados irão conter repetições e algum algoritmo de eliminação de duplicatas deve ser implementado. Isto ocorre devido ao fato de a estrutura ser um grafo onde diversas árvores podem compartilhar diversos nós.

Desta forma, iremos reformular a consulta por intervalo de tempo para obter uma forma mais eficiente de execução tirando proveito da estrutura da TR-Tree. A consulta por intervalo de tempo será formulada então da seguinte forma: "encontrar todos os MBRs que interceptam o MBR R e foram mortos no intervalo de tempo [t1, t2) ou estão vivos no tempo t2". Embora esta consulta produza o mesmo resultado, ela será efetuada de forma mais eficiente e não irá encontrar resultados duplicados uma vez que:

- entradas mortas não são copiadas após uma divisão de versão, logo, não são duplicadas;
- no tempo t2 existe apenas uma entrada viva para cada MBR.

Em seguida apresentaremos uma demonstração formal de que esta idéia gera um algoritmo que não segue percursos duplicados e nem retorna resultados duplicados e que, conseqüentemente, o algoritmo possui melhor desempenho do que a abordagem ingênua e retorna o resultado correto. Inicialmente iremos ressaltar algumas propriedades da TR-Tree que irão garantir o bom funcionamento do algoritmo de busca.

Para facilitar o entendimento desta demonstração, encontra-se na Figura 16 que mostra uma seqüência de divisões de versão em um nó e o que acontece com suas entradas vivas e mortas. Este comportamento após a operação de divisão de versão é que rege as propriedades que serão enumeradas em seguida.



Esta figura representa a situação de uma base de dados gerada com 1000 objetos iniciais e 100 versões. É mostrado nesta figura a evolução dos objetos contidos no nó 28, mostrando-se a seqüência de divisões de versão deste nó. Na nomenclatura adotada, os nós 116, 130 e 249 são sucessores do nó 28. Conseqüentemente, o nó 28 é antecessor destes nós. Repare que o objeto com ID=1 é um objeto vivo desde o tempo 1 e foi copiado por todos os nós. Repare também como os objetos já removidos não são copiados para o sucessor, como por exemplo o nó com ID=498 no nó 130. Estas propriedades que são de fundamental importância para o algoritmo de busca por intervalo de tempo, bem como o algoritmo de junção espaçotempoal

Figura 16. Seqüência de divisões de versão de um nó.

Assume-se a seguinte suposição: seja E uma entrada com tempo de vida [n,m), contida nos nós X_1, X_2, ..., X_k da árvore A, e sejam os intervalos $[n_1,m_1)$, $[n_2,m_2)$, ... $[n_k,m_k)$ os respectivos tempos de vida de cada nó. Obviamente, cada $n_i < m_i$ para $1 \leq i \leq k$. Seguem então as seguintes propriedades:

P1. $m_i = n_i + 1$, para $1 \leq i < k$

Justificativa: Entradas são duplicadas apenas através da operação de divisão de versão. Neste caso, todas as entradas vivas são inseridas em nós criados obrigatoriamente no tempo corrente, e o nó que sofreu a divisão de versão é marcado como morto. Nenhuma entrada que foi duplicada pode ser inserida em um nó que não tenha sido criado no tempo corrente, ou seja, em algum tempo diferente do tempo em que o nó que continha a entrada morreu. Desta forma, ao longo do tempo de vida de um dado objeto ou nó, em cada instante, um e apenas um nó possui a entrada E para o referido objeto ou nó. Iremos assim chamar de sucessor de X_i para E ao nó que contém a entrada de E após a morte de X_i, ou seja, X_{i+1}. Estas relações entre os nós pode ser

vista na Figura 16. Note que as entradas mortas em X_i não possuem sucessores, pois não serão copiadas para X_{i+1} quando X_i morrer.

P2. $n \in [n_1, m_1)$.

Justificativa: As entradas, ao serem criadas, devem ser inseridas em nós que já tenham sido criados e que estejam vivos.

P3. Se E está morta, então $m \in [c_k, m_k)$. Se E está viva, então X_k também está vivo e $m \geq c_k$.

Justificativa: Se a entrada E morreu no tempo m, então ela não será mais copiada quando o nó que a contém morrer. Além disso, entradas mortas não são copiadas. Logo, se E morreu, morreu após ter sido inserida em X_k e antes de X_k morrer, pois caso contrário não pertenceria a X_k. Se E não morreu, então X_k também não, pois caso contrário E deveria ter sido copiada para um outro nó criado no tempo de morte de X_k.

P4. A entrada E está viva dentro dos nós $X_1, X_2, ..., X_{k-1}$.

Justificativa: Se a entrada E não estiver viva dentro de algum destes nós não seria duplicada para o próximo e a cadeia seria quebrada, uma vez que esta cadeia é gerada pela operação de divisão de versão e esta operação não duplica entradas mortas.

Juntando estas propriedades P1, P2, P3 e P4, podemos afirmar que:

P5. $X_1, X_2, ..., X_{k-1}$ estão mortos e contêm a entrada E viva

P6. Se X_k está morto, então a entrada E está morta

P7. Se X_k está vivo, então a entrada E pode estar viva ou morta

Estas três novas propriedades são agora suficientes para provar que o algoritmo relata a entrada E apenas uma vez. A propriedade P5 garante que a entrada E não vai ser relatada ao percorrermos os nós $X_1, X_2, ..., X_{k-1}$. As propriedades P6 e P7 restantes garantem que a entrada E será com certeza relatada ao percorrermos o nó X_k. Note que não fizemos nenhuma restrição ao tipo de entrada E e, portanto, estes conceitos valem

para qualquer entrada na estrutura. Isto mostra que entradas na TR-Tree não são percorridas mais de uma vez durante o algoritmo de busca por intervalo de tempo.

Falta agora mais um passo da prova, que é o de mostrar que todas as entradas que satisfazem o predicado de consulta são percorridas, pois mostramos apenas que o resultado não contém duplicatas, mas não que o resultado é completo.

Para tal, precisamos apenas mostrar que o nó X_k deve ser visitado uma e somente uma vez durante a busca, já que a entrada E é percorrida somente neste nó. Iremos demonstrar este fato utilizando-se uma indução no nível dos nós na árvore com hipótese de que todos os nós da estrutura que satisfazem o predicado da consulta são percorridos uma e somente uma vez pela consulta. Seja N um nó qualquer na estrutura da TR-Tree.

O passo inicial da indução ocorre quando N é um nó raiz. Caso N satisfaça o predicado de consulta, N será visitado pela consulta e será visitado apenas uma vez, pois os nós raízes da árvore são visitados apenas uma vez no laço "para cada nó raiz que satisfaz o predicado da consulta" do algoritmo.

Para o passo seguinte da indução, iremos assumir que a hipótese de indução vale para todos os nós em um nível *n* na estrutura da TR-Tree. Neste nível então, os nós que satisfazem o predicado da consulta são visitados uma e somente uma vez pelo algoritmo de busca. As entradas neste nível da árvore podem encontrar-se duplicadas, mas sempre respeitando as propriedades P1, ..., P7. Desta forma, como foi mostrado anteriormente, podemos verificar que as entradas neste nível *n* para os nós no nível *n+1* ou para os registros de índice contendo os objetos espaciais, que satisfazem o predicado da consulta, só irão ser visitadas uma única vez. Como a hipótese de indução vale para o nível *n*, os nós deste nível que satisfazem o predicado de consulta serão visitados uma e apenas uma vez pelo algoritmo de busca, todas as entradas nestes nós que também satisfazem o predicado de consulta serão necessariamente visitadas e como mostrado acima, também apenas uma vez. Conseqüentemente fica mostrado que os nós no nível *n+1* também serão visitados uma e somente uma única vez.

Com esta indução consegue-se mostrar que todos os nós que satisfazem o predicado de consulta são visitados uma e somente uma única vez pelo algoritmo de busca. Juntando esta prova com a anterior de que cada entrada é visitada apenas uma

vez pelo algoritmo de busca podemos afirmar que os objetos espaciais que satisfazem o predicado da consulta tanto espacial quanto temporalmente são relatados apenas uma vez.

Note que esta prova é ainda mais importante do que apenas mostrar que os objetos, com este algoritmo de busca, não são relatados duplicados. Esta prova mostra como o algoritmo é eficiente, uma vez que não percorre nós e entradas desnecessariamente, reduzindo assim drasticamente o número de acessos a disco para a realização da consulta.

A seguir encontram-se listados os algoritmos que efetuam a busca por intervalo de tempo detalhada nesta seção.

```
Procedimento BuscaIntervalo( R: MBR; [t1,t2): Intervalo )
Inicio
  Para cada nó raiz N no vetor de nós raízes cujo t_nasc ∈ [t1,t2) exceto o último
faça
     BuscaIntervaloEntradasMortas( N, R, [t1,t2) )

  // Para o último nó raiz
  BuscaIntervaloTodasEntradas( N, R, [t1,t2) )
Fim
```

Algoritmo 11. Algoritmo de Busca por Intervalo de Tempo – Procedimento BuscaIntervalo

```
Procedimento BuscaIntervaloEntradasMortas( N: Nó; R: MBR; [t1,t2): Intervalo )
Inicio
  Se N é folha Então
    Para cada entrada E em N faça
      Se E E.MBR ∩ R E E.t_morte ∈ [t1, t2) Então
        Retornar E como resultado
  Senão
    Para cada entrada E em N faça
      Se E.MBR ∩ R E E.t_morte ∈ [t1, t2) Então
        BuscaIntervaloEntradasMortas( E.ponteiro, R, [t1,t2) )
Fim
```

Algoritmo 12. Algoritmo de Busca por Intervalo de Tempo – Procedimento BuscaIntervaloEntradasMortas

```
Procedimento BuscaIntervaloTodasEntradas( N: Nó; R: MBR; [t1,t2): Intervalo )
Inicio
  Se N é folha Então
    Para cada entrada E em N faça
      Se E E.MBR ∩ R E [E.t_nasc,E.t_morte) ∩ [t1, t2) Então
        Retornar E como resultado
  Senão
    Para cada entrada E em N faça
      Se E.MBR ∩ R E E.t_morte ∈ [t1, t2) Então
        BuscaIntervaloEntradasMortas( E.ponteiro, R, [t1,t2) )
      Senão Se E.MBR ∩ R E [E.t_nasc,E.t_morte) ∩ [t1, t2) Então
        BuscaIntervaloTodasEntradas( E.ponteiro, R, [t1,t2) )
Fim
```

Algoritmo 13. Algoritmo de Busca por Intervalo de Tempo – Procedimento BuscaIntervaloTodasEntradas

Das propriedades P5, P6 e P7 surge a idéia do algoritmo de busca por intervalo de tempo (*BuscaIntervalo*) mostrado acima. O algoritmo consiste em percorrer a estrutura, relatando as entradas que interceptam o retângulo de busca R mas, nos nós que já morreram, listando apenas as entradas mortas (invalidando a propriedade P5 e evitando o acesso a E pelos nós X_1, X_2, ..., X_{k-1}), pois as entradas vivas irão ser listadas mais adiante nos sucessores deste nó (Algoritmo *BuscaIntervaloEntradasMortas*). Nos nós que estão vivos no final do intervalo de busca (ou seja, que estão realmente vivos ou morreram em um instante após o fim do intervalo de busca) devemos relatar todas as entradas (Algoritmo *BuscaIntervaloTodasEntradas*), pois não iremos investigar os sucessores de deste nó (propriedades P6 e P7).

4.3. Junção Espaçotemporal

A consulta de junção espaçotemporal por intervalo de tempo recebe como parâmetros um retângulo de busca *R* e um intervalo de tempo de busca $[t_1, t_2)$. Uma junção espaçotemporal por instante de tempo pode ser vista como uma junção espaçotemporal por intervalo de tempo com intervalo de tempo de busca $[t_1, t_1+1)$. Esta consulta retorna todos os pares de objetos espaciais que possuem interseção entre si e com o retângulo de busca R e cujo intervalo de tempo de vida intercepta-se com o intervalo de tempo de busca.

Da mesma forma que na busca por intervalo de tempo, uma estratégia ingênua seria a de percorrer os nós raízes das duas estruturas e efetuar as junções espaçotemporais em cada árvore. É importante notar que a junção espaçotemporal por instante de tempo irá um e somente um par de árvores. O algoritmo para junção de cada árvore seria bastante semelhante ao da R*-Tree proposto em BRINKHOFF et al. (1993)

levando-se ainda em conta o tempo. Este algoritmo levaria a um número excessivo de junções de R*-Trees, que por si só já é um problema complexo, e ainda acarretaria no problema de retornar pares de entradas duplicados. O problema é um pouco mais grave, pois devemos lembrar que uma entrada duplicada para um nó irá significar que este nó será visitado mais de uma vez. Se este problema ocorresse somente no nível dos nós folha, o única medida necessária seria a de um mecanismo de eliminação de duplicatas, mas ele se verifica também em nós internos da árvore, o que implica em um número excessivo de acessos a disco ou então um mecanismo de eliminação de duplicatas de pares de entradas por nível da árvore com a utilização de um algoritmo de busca em largura. A Figura 17 ilustra este problema.

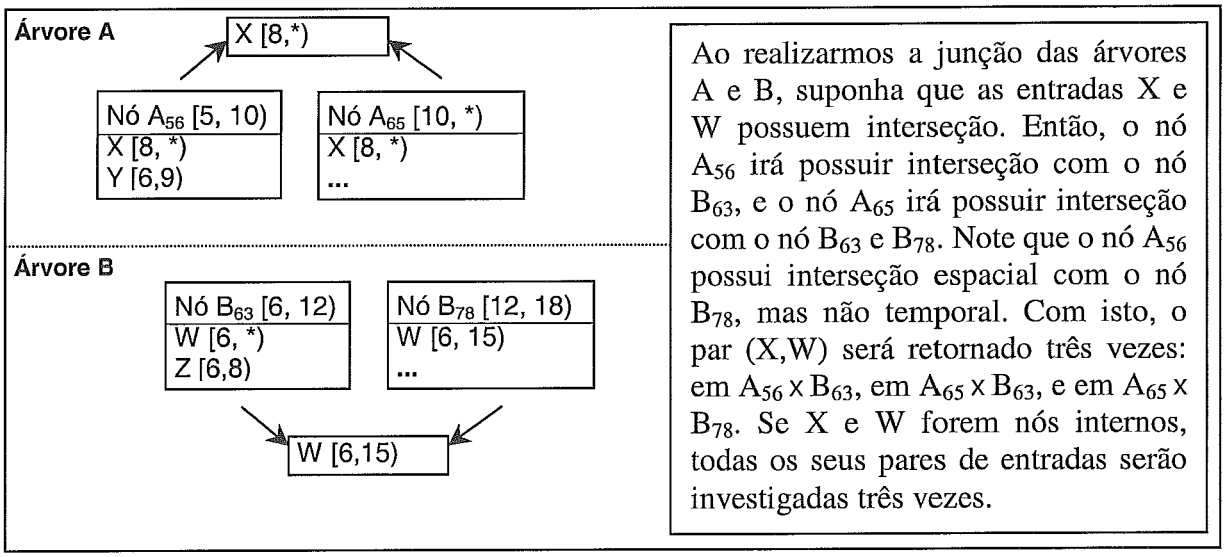


Figura 17. Junção Espaçotemporal – problema dos pares duplicados

Pior do que isso, se elas forem relatados em outro nó para o qual também existe uma entrada duplicada, elas serão relatadas duas vezes neste outro nó também, e assim por diante em cada nível da árvore até a raiz. Claramente esta não é uma boa abordagem. Um método mais eficiente deve ser buscado, pois senão seria impraticável a implementação de tais algoritmos de junção espaçotemporal.

Seguindo a mesma idéia da busca por intervalo de tempo, podemos estabelecer um algoritmo com estratégia semelhante para a junção e que não percorra pares duplicados de entradas. Vamos supor que o algoritmo investigue cada par de nós que possuem interseção espacial e temporal apenas uma vez. Fatalmente, como vimos na Figura 17, haverá pares que aparecerão mais de uma vez. Devemos, portanto, estabelecer um critério que permita decidir se um determinado par deve ou não ser

listado. Uma boa abordagem seria listar os objetos no último tempo de sua interseção, pois como vimos anteriormente, embora a árvore duplique entradas, apenas uma delas, a do nó que não possui sucessores, pode marcar a sua morte. A transformação da consulta para junção espaçotemporal análoga à que foi feita para a consulta por intervalo de tempo seria: "encontre todos os pares de MBRs com sobreposição que possuem o tempo final de sua interseção durante o intervalo da consulta, ou que estejam vivos no tempo final deste intervalo".

Usando a consulta acima podemos então estabelecer critérios para decidir se dois pares de entradas candidatas devem ou não ser relatadas quando encontradas. Chamamos de pares candidatos de entradas os pares de entradas que possuem interseção espacial e temporal, e sempre que mencionarmos se um nó ou entrada está morto ou vivo, estamos nos referindo ao seu estado no instante final do intervalo da consulta.

1) Se os nós onde os pares candidatos de entradas estão são nós vivos no final do intervalo da consulta, temos que estes dois nós não possuem sucessores (ou seus sucessores não serão visitados, pois foram criados após o fim do intervalo da consulta), logo todos os pares candidatos de entradas devem ser relatados neste momento, pois não iremos ter uma nova chance de fazê-lo.

2) Se um dos nós está morto, e o outro está vivo no tempo final da consulta, devemos relatar apenas os pares candidatos em que as entradas que pertencem ao nó morto também estejam mortas, pois os pares candidatos de entradas que contém entradas vivas do nó morto irão aparecer de novo nos sucessores deste nó para cada uma destas entradas, e certamente cada um destes sucessores irá possuir interseção com o nó que está vivo.

3) Se ambos estão mortos, teremos um pouco mais de trabalho: os pares candidatos em que estão ambas as entradas vivas certamente não devem ser relatadas, pois o mesmo irá aparecer novamente como par candidato nos respectivos sucessores. Da forma análoga, os pares candidatos que possuem apenas entradas mortas também devem ser relatados, pois esta é última vez que aparecerão na árvore (não possuem sucessores). Porém, os pares de entradas em que aparece uma entrada morta e uma entrada viva apresentam a seguinte dificuldade: seja A o nó que contém a entrada viva e B o nó que contém a entrada morta. Se A morreu antes de B, o sucessor de A

será comparado com B, e nele iremos novamente encontrar o mesmo par candidato de entradas. Note que este problema não irá surgir se os dois nós tiverem morrido no mesmo tempo, pois nenhum dos sucessores será comparado com o outro nó. Assim, não devemos listar os pares de entradas em que uma está viva e a outra está morta se o nó que contém a entrada viva morreu antes do nó que contém a entrada morta.

Os algoritmos que executam a junção espaçotemporal e que refletem a idéia explicitada acima encontram-se listados abaixo.

```
Procedimento JuncaoEspaçotemporal(Ta, Tb: TR-Tree; I: Intervalo; R: MBR)
Inicio

  A = EncontrarRaiz(Ta, I.Inicio)
  B = EncontrarRaiz(Tb, I.Inicio)
  ProximoA = False
  ProximoB = False
  Faça
    Se ProximoA Então
      A = ProximoRaiz(Ta)
    Se ProximoB Então
      B = ProximoRaiz(Tb)

    ParesObjetos = ParesObjetos U JuncaoEspacoTemporalNo(A, B, I, R)

    ProximoA = A.t_morte <= B.t_morte
    ProximoB = A.t_morte >= B.t_morte

  Enquanto not Vivo(A, I.Fim) or not Vivo(B, I.fim)

  Retornar ParesObjetos como resultado

Fim
```

Algoritmo 14. Algoritmo de Junção Espaçotemporal – Procedimento JunçãoEspaçotemporal

```
Procedimento JuncaoEspacotemporalNo( noA, noB: Nó; R: MBR; I: Intervalo ): ListaEntradas
Inicio
  Saida = ListaVazia
  i = 1; j = 1
  listaA = lista de entradas de noA que possuem interseção espacial com R e
           temporal com I, ordenadas na menor cordenada x e na menor coordenada y
  listaB = lista de entradas de B que possuem interseção espacial com R e
           temporal com I, ordenadas na menor cordenada x e na menor coordenada y
  Enquanto i ≤ Tamanho(listaA) and j ≤ Tamanho(listaB) Faça
    Se listaA[ i ].min.x < listaB[ j ].min.x Então
      Saída = Saída U LaçoInterno(listaA[i], j, listaB, I.Final, False, noA, noB)
      i = i + 1
    Senão
      Saida = Saida U LaçoInterno(listaB[j], i, listaA, I.Final, True, noB, noA)
      j = j + 1
  Retornar Saida
Fim
```

Algoritmo 15. Algoritmo de Junção Espaçotemporal – Procedimento JuncaoEspacotemporalNo

```
Procedimento LaçoInterno( E: Entrada; NaoMarcado: Inteiro; Lista: ListaEntradas;
                          FinalInt: Tempo; Trocar: Boolean; noA, noB: Nó): ListaEntradas
Inicio
  Saída = ListaVazia
  k = NaoMarcado
  Enquanto k ≤ Tamanho(Lista) and Lista[ k ].min.x < E.max.x Faça
    Se E.min.y ≤ Lista[ k ].max.y and
       E.max.y ≥ Lista[ k ].min.y and
       E.intervalo ∩ Lista[ k ].Intervalo ≠ ∅ and
       ListaPar( E, Lista[ k ], noA, noB, FinalInt ) Então
      Se Trocar = False Então
        Saída = Saída U Par( E, Lista[ k ] )
      Senão
        Saída = Saida U Par( Lista[ k ], E )
    k = k + 1
  Retornar Saida
Fim
```

Algoritmo 16. Algoritmo de Junção Espaçotemporal – Procedimento LaçoInterno

```
Função ListaPar(eA, eB: Entrada; noA, noB: No; FinalInt: Tempo): Boolean
Inicio
  Se Vivo(noA, FinalInt) and Vivo(noB, finalInt) Então
    Retornar True
  Senão Se (Vivo(noA, FinalInt) and Morto(noB, finalInt) and Morta(eB, FinalInt)) or
           (Morto(noA, FinalInt) and Vivo(noB, FinalInt) and Morta(eA, FinalInt)) Então
    Retornar True
  Senão // ambos os nós são mortos.
    Se noA.t_morte = noB.t_morte and
       (Morta(eA, FinalInt) or Morta(eB, FinalInt)) Então
      Retornar True
    Senão Se (noA.t_morte < noB.t_morte and Morta(eA, FinalInt)) or
             (noA.t_morte > noB.t_morte and Morta(eB, FinalInt)) Então
      Retornar True
    else
      Retornar False
Fim
```

Algoritmo 17. Algoritmo de Junção Espaçotemporal – Função ListaPar

Alguns comentários sobre estes algoritmos ainda podem ser tecidos:

- A variável "trocar" no procedimento *LaçoInterno* indica se a ordem dos pares deve ser trocada e é utilizada apenas para que não seja necessário escrever duas funções quase idênticas.

Iremos agora tentar provar a corretude do algoritmo proposto para o problema de junção espaçotemporal. Para tal, suponha que A e B sejam dois nós com pares de entradas candidatas E e F. Isto significa que E está presente no nó A e F está presente no nó B e que $E \cap F \neq \emptyset$, ou seja, os MBRs de E e F possuem interseção não vazia. Queremos mostrar que os dois nós A e B só são comparados pelo algoritmo uma e somente uma vez, o que significa dizer que as entradas E e F só serão testadas em conjunto uma e somente uma única vez pelo algoritmo de junção espaçotemporal.

Inicialmente podemos mostrar que o algoritmo irá retornar todos os pares de entradas que possuem interseção entre si devido ao fato do mesmo percorrer todos os nós raízes das duas TR-Trees de forma sincronizada e depois, para cada árvore, efetuar uma busca em profundidade passando por todas as entradas que possuam interseção. Desta forma garante-se que o resultado é completo, mas não está mostrado que o resultado é minimal, ou seja, que não contém duplicatas.

Para mostrar que o resultado não contém duplicatas iremos utilizar o mecanismo de indução para prova de hipóteses. A indução será feita no nível da árvore onde se encontram os nós A e B. Note que não necessariamente A e B encontram-se no mesmo nível da árvore, mas iremos assumir isto para simplificar a demonstração. Uma prova para níveis diferentes seria bastante semelhante.

O passo inicial da indução ocorre nos nós raízes da TR-Tree. Neste nível não há sobreposição de tempos dos nós, ou seja, não há interseção temporal entre os diversos nós raízes de uma TR-Tree. Devido a este fato, o laço inicial do algoritmo que percorre os nós raízes nunca compara o mesmo par de nós mais de uma vez. Pode acontecer de um nó ser comparado a diversos outros, mas nunca um par ser comparados mais de uma vez. Isto pode ser visto na Figura 18.

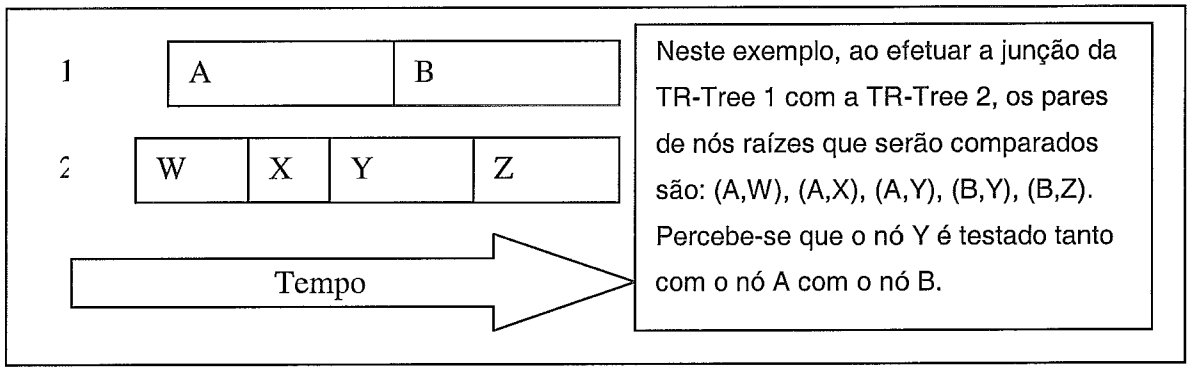


Figura 18. Duplicação de pares de entradas nos nós raízes.

Suponha então que a hipótese de indução valha para o nível *n* das TR-Trees, ou seja, os pares de nós encontrados no nível *n* de ambas as TR-Trees serão visitados pelo algoritmo de junção espaçotemporal uma e somente uma única vez. Queremos mostrar que a hipótese vale também para o nível *n+1*. Para tal suponha novamente que A e B sejam dois nós no nível *n* da árvore e E e F dois pares de entradas candidatas.

Note que o algoritmo desce de nível em uma árvore da TR-Tree pelas três condições citadas anteriormente que encontram-se aqui resumidas para dar mais clareza a demonstração:

Condição 1: Se A e B são nós vivos no final do intervalo da consulta, logo todos os pares candidatos de entradas devem ser relatados

Condição 2: Se um dos nós está morto, e o outro está vivo no tempo final da consulta, devemos relatar apenas os pares candidatos em que as entradas que pertencem ao nó morto também estejam mortas

Condição 3: Se ambos os nós A e B estão mortos, os pares candidatos com ambas as entradas vivas não são relatados, os pares candidatos contendo ambas entradas mortas são relatados. Porém nos pares de entradas em que aparece uma entrada morta e uma entrada viva, não devemos listar os pares de entradas se o nó que contém a entrada viva morreu antes do nó que contém a entrada morta.

Iremos dividir esta demonstração em três casos distintos:

1. Se ambos os nós estão vivos no final do intervalo de consulta. Neste caso o algoritmo irá visitar o par de entradas E e F e descer nos nós X e Y apontados por E e F respectivamente, ou retornar E e F caso os nós A e B sejam folhas. Os nós A e B não têm sucessores uma vez que encontram-se vivos. O algoritmo testaria X e Y apenas se tivesse descido por E e F por algum par de antecessores de A e B que contêm E e F. Existem aqui três possibilidades:

   1.1. O par de nós é formado por dois antecessores quaisquer de A e B. Todos os antecessores de A e B encontram-se necessariamente mortos, logo apenas a condição 3 tem possibilidade de ser aplicada. As entradas E e F eram necessariamente vivas nos nós antecessores a A e B pois caso contrário não teriam sido copiadas na operação de divisão de versão e conseqüentemente não estariam presentes em A e B. Para nós mortos e entradas vivas, a condição 3 do algoritmo não é satisfeita. Logo, neste caso as entradas E e F não seriam relatadas pelo algoritmo.

   1.2. O par de nós sendo testado contém um antecessor qualquer de A e o nó B. Todos os antecessores de A encontram-se necessariamente mortos. Estamos

testando então um nó morto (antecessor de A) com um nó vivo (B). Apenas a condição 2 tem possibilidade de ser satisfeita, mas a entrada E encontra-se viva em todos os antecessores de A, o que invalida também a condição 2 que só testa entradas mortas do nó que encontra-se morto.

1.3. O par de nós sendo testado contém um antecessor qualquer de B e o nó A. A demonstração é a mesma do caso anterior uma vez que as condições são as mesmas.

Desta forma, se ambos os nós A e B encontram-se vivos, o algoritmo desce pelas entradas E e F apenas nestes nós, pois os mesmos não possuem sucessores e as condições de descida não são satisfeitas nos antecessores.

2. Se ambos os nós A e B estão mortos no tempo final do intervalo de consulta. Aqui podem ocorrer 3 casos distintos:

2.1. Ambas as entradas E e F estão vivas. Estas entradas não são relatadas pois nenhuma das condições é satisfeita. Repare que as mesmas serão com certeza relatadas em algum par de nós sucessores de A e B, pois serão copiadas para os mesmos na operação de divisão de versão.

2.2. Ambas as entradas estão mortas. As entradas serão relatadas pelo algoritmo através da condição 3. Estas entradas não estarão presentes nos sucessores de A e B, logo tenho apenas que me preocupar se estão sendo relatadas nos antecessores de A e B. Nos antecessores, estas entradas estão vivas. Podem acontecer novamente 3 casos:

2.2.1. Estarão sendo percorridos dois antecessores de A e B. Neste caso os antecessores encontram-se necessariamente mortos e as entradas E e F encontradas nos mesmo encontram-se vivas. Nenhuma das condições é satisfeita para este caso.

2.2.2. Estarão sendo percorridos um antecessor de A (A') com o nó B. Todos os antecessores de A encontram-se mortos e a entrada E encontrada nos mesmos encontra-se viva. O outro nó, o nó B, encontra-se morto com a entrada F morta. A condição 3 do algoritmo só irá relatar as entradas E e F em A' e B caso A' não tenha morrido antes de B. Este caso não pode

ocorrer, pois senão A e B não estariam sendo testados pelo algoritmo por não possuírem interseção temporal, logo E e F também não possuiriam interseção temporal e assumimos no início da demonstração que E e F são entradas candidatas. Esta afirmação só vale porque E e F estão mortas. A Figura 19 elucida melhor este problema.

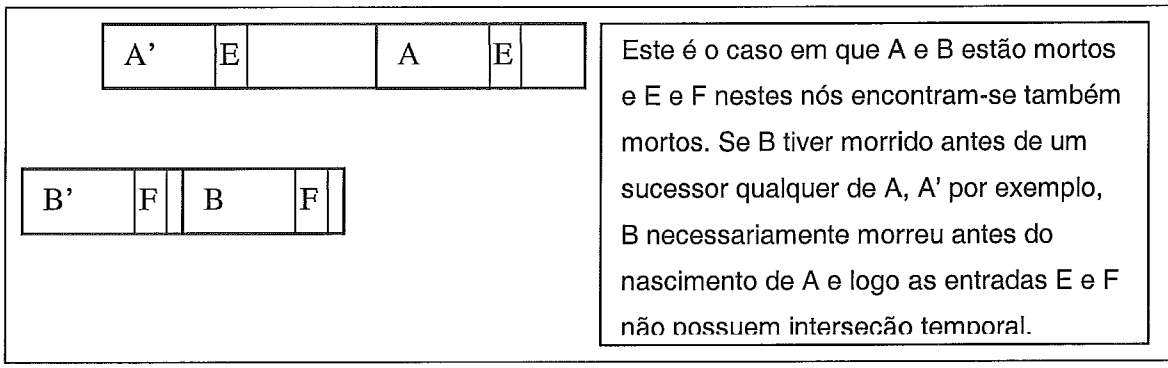


Figura 19. Elucidando o Caso 2.2.2.

2.2.3. Estarão sendo percorridos um antecessor de B (B') com o nó A. A demonstração é a mesma do caso anterior uma vez que as condições são as mesmas.

2.3. Uma entrada está morta e a outra viva. Suponha, sem perda de generalidade, que o nó que contém a entrada viva E é o nó A, logo o nó B contém a entrada morta F. A condição 3 só é satisfeita se A morreu após a morte de B. Neste caso, só preciso verificar se está sendo percorrido algum antecessor de A com o nó B ou com algum antecessor de B, uma vez que o as entradas mortas em B não são copiadas para seus sucessores. Novamente temos três casos:

2.3.1. Estarão sendo percorridos dois antecessores de A e B. Neste caso, os antecessores estão mortos e as entradas E e F vivas. Já foi mostrado anteriormente que nenhuma condição é satisfeita para este caso.

2.3.2. Estarão sendo percorridos um antecessor de A (A') com o nó B. Neste caso, o antecessor de A, o nó A', está morto e a entrada E contida nele encontra-se viva. Estamos então testando dois nós mortos, A' e B, um com uma entrada viva (A') e outro com uma entrada morta (B). Este caso contém as mesmas condições que o caso 2.2.2, logo as entradas E e F não serão retornadas.

2.3.3. Estarão sendo percorridos um antecessor de B (B') com o nó A. Assim como o caso anterior, este caso contém as mesmas condições que o caso 2.2.3.

3. Se um dos nós está vivo e o outro está morto no tempo final do intervalo de consulta. Suponha, sem perda de generalidade, que A encontra-se vivo e B morto. O algoritmo somente relata as entradas em B que estão mortas. Estas entradas não aparecerão nos sucessores de B pois não serão copiadas na operação de divisão de versão. Logo, somente preciso me preocupar em pesquisar se as entradas estão sendo duplicadas em algum antecessor de A e de B. Podem ocorrer três casos:

   3.1. Estarão sendo percorridos dois antecessores de A e B. Neste caso, os antecessores estão mortos e as entradas E e F vivas. Já foi mostrado anteriormente que nenhuma condição é satisfeita para este caso.

   3.2. Estarão sendo percorridos um antecessor de B com o nó A. Neste caso, todos os antecessores de B encontram-se mortos e a entrada F pertencente a B encontra-se viva nestes nós, o que invalida a única condição que ainda poderia se tornar possível, a condição 2.

   3.3. Estarão sendo percorridos um antecessor de A com o nó B. Todos os antecessores de A encontram-se necessariamente mortos e as entradas em A nos antecessores encontram-se necessariamente vivas. Se a entrada F encontrar-se viva em B, a condição 3 encontra-se invalidada. Caso contrário, a condição 3 só pode ser satisfeita caso o nó que contém a entrada viva, no caso algum antecessor de A, tiver morrido após o nó que contém a entrada morta, o nó B. Este caso é inviável pelo mesmo problema apontado em 2.2.2 melhor elucidado graficamente na Figura 19.

Esgotando-se todas as possibilidades podemos dizer que a hipótese de indução também vale para o nível $n+1$ da árvore, finalizando assim a prova por indução no nível da árvore onde se encontra o par de nós.

4.4. Sumário

Neste capítulo, portanto, vimos em detalhes como é a estrutura da TR-Tree. Os algoritmos de busca, inserção e remoção foram apresentados, bem como o de junção espaçotemporal. Além disso, provamos formalmente que os algoritmos de busca e junção por intervalo de tempo retornam todas as entradas ou os pares de entradas do conjunto resposta uma apenas uma vez. No capitulo seguinte, iremos ver os resultados dos testes efetuados comparando a TR-Tree com outras estruturas espaçotemporais implementadas.

No próximo capítulo será feita uma avaliação experimental de desempenho dos algoritmos propostos neste capítulo através de comparações feitas com outros métodos de acesso implementados.

5. Análise Experimental

Neste capítulo iremos mostrar a eficiência da estrutura de indexação proposta nesta dissertação através de uma análise experimental comparando o desempenho de seus algoritmos com o desempenho dos algoritmos de alguns métodos de acesso propostos na literatura.

A Seção 5.1 comenta alguns detalhes de implementação e o porquê da escolha dos métodos de acesso implementados e testados. A Seção 5.2 passeia pelo ambiente de desenvolvimento e dos testes experimentais. A Seção 5.3 fala sobre a geração dos conjuntos de dados utilizados bem como sobre as consultas efetuadas. A Seção 5.4 justifica a escolha do tamanho da página que irá ser utilizado em nossos testes. As Seções 5.7, 5.8, 5.9 e 5.10 mostram os resultados das consultas de seleção por instante de tempo, por intervalo de tempo e das consultas de junção espaçotemporal por instante de tempo e por intervalo de tempo, respectivamente.

5.1. Implementação

Como não poderia ser diferente, os algoritmos propostos no Capítulo 4 para a TR-Tree foram todos implementados. Para que pudéssemos testar o desempenho dos mesmos foi necessário implementar os principais algoritmos de alguns métodos de acesso espaçotemporais encontradas na literatura citadas no Capítulo 3.

Implementamos apenas os métodos de acesso espaçotemporais com suporte a tempo de transação (Seção 3.3.2) uma vez que a TR-Tree é um método de acesso com esta característica. Desta lista de métodos de acesso eliminamos a OLQT e a MLQT devido ao fato de lidarem com Quadtrees Lineares não servindo portanto para efeito de comparação. Não implementamos também a MR-Tree, a 3D R-Tree e a PPR-Tree. A MR-Tree utiliza os mesmos princípios da HR-Tree e esta última encontra-se muito bem vista na literatura, inclusive como o método de acesso mais promissor para o armazenamento de dados espaçotemporais. A 3D R-Tree possui também sua correspondente 2+3D R-Tree que resolve o problema do tempo atual crescente e indeterminado. A PPR-Tree é uma estrutura muito semelhante à TR-Tree e, embora tenha sido criada posteriormente, utiliza a técnica de fusão de nós quando um nó

encontra-se abaixo da capacidade mínima. A PPR-Tree também não implementa a reinserção forçada de entradas dos algoritmos da R*-Tree.

Como a RT-Tree já possui um fraco desempenho nos algoritmos de consulta e um elevado tempo de criação de seus índices, como foi mostrado em ZIMBRÃO et al. (2000), esta não foi utilizada nesta dissertação. Visivelmente a RT-Tree não é uma boa abordagem, pois não leva em conta o tempo na organização da árvore. A 3D R-Tree já é uma estrutura melhorada para tal problema.

Com isso, foram implementadas e testadas além da TR-Tree, a 2+3D R-Tree e a HR-Tree. Os algoritmos de busca por instante de tempo e por intervalo de tempo foram implementados em todas as estruturas para uma análise de desempenho. O algoritmo de junção espaçotemporal não foi implementado para a HR-Tree pois seria bastante ineficiente. A HR-Tree não armazena a informação temporal nos nós de suas árvores e um algoritmo de junção espaçotemporal por intervalo de tempo deveria percorrer duas a duas cada Hilbert R-Tree (pertencentes ao intervalo de tempo de junção) efetuando junções espaciais. Visivelmente esta não é uma boa abordagem e como não existe nenhuma outra abordagem proposta na literatura, resolvemos por não implementar nenhum algoritmo de junção espaçotemporal para a HR-Tree.

Para o caso da 2+3D R-Tree, um algoritmo óbvio para a realização de junções espaçotemporais é proposto e implementado. Em sua essência, a estrutura de cada uma das duas árvores que compõem a 2+D R-Tree é idêntica à R-Tree original: apenas uma delas possui duas dimensões (a que armazena os objetos ainda vivos no tempo corrente) e a outra possui três dimensões. Realizar uma junção sobre duas 2+3D R-Trees é portanto semelhante a realizar quatro junções em R-Trees: uma junção sobre um par de árvores com duas dimensões, outra sobre um par de árvores com três dimensões, e finalmente duas junções sobre árvores com duas e três dimensões. Apenas o último caso possui uma particularidade que o torna diferente do algoritmo original: realizar a junção sobre árvores com número de dimensões diferentes. Porém, cabe lembrar ao leitor que o número de dimensões é diferente apenas no que toca ao algoritmos estruturais da árvore, ou seja, de inserção e divisão de nós. Implicitamente temos, na árvore de duas dimensões, a informação da terceira dimensão armazenada também. Esta dimensão corresponde exatamente ao tempo de transação: as entradas nesta árvore possuem um campo com o seu tempo de criação, e sabe-se que as mesmas estão todas vivas. Logo, a

dimensão que faltava será sempre representada por [t_nasc,*), onde t_nasc é o tempo de criação da entrada (no caso dos nós, o tempo de criação da entrada mais antiga contida neste nó). Assim, estamos equiparados ao caso de realizar junções em árvores de MBRs de três dimensões. Cabe observar ainda que a estrutura da 2+3D R-Tree não duplica entradas, de forma que, embora tenhamos de realizar quatro junções completas, não é necessário eliminar pares de resposta duplicados.

5.2. Ambiente de Desenvolvimento

Algumas máquinas foram utilizadas para a implementação desta dissertação, mas a máquina em que a análise experimental foi executada foi uma Intel Pentium III de 700 MHz e 256 MBytes de memória RAM e 20 GBytes de disco rígido. Esta máquina, nos momentos de execução dos testes de desempenho não encontrava-se disponível para outros usuários, pois estávamos interessados em medir também os tempos de consulta. A linguagem de programação utilizada em toda a implementação foi o C++ devido à sua comprovada velocidade e facilidades como o recurso de orientação a objetos, por exemplo. O ambiente de desenvolvimento utilizado foi o Conectiva Linux Server 6.0 com compilador gcc 2.95.

Para cada base de dados utilizamos um cache LRU de 97 páginas de 4 Kbytes. A justificativa para a escolha deste tamanho de página encontra-se na Seção 5.4. Um cache adicional do tamanho da altura das árvores foi utilizado para armazenar o caminho percorrido nos algoritmos. Este cache garante que uma subida de nível na árvore não irá gerar uma acesso adicional a disco independentemente do cache LRU.

Como a máquina utilizada em nossos testes possui bastante memória (256 MBytes), os métodos de acesso praticamente cabiam na memória principal, exceto a HR-Tree, e o cache do sistema operacional estava tendo altíssima influência nos resultados dos tempos de consulta. Tentamos resolver este problema da seguinte maneira: inicialmente, executamos um pequeno programa que mediu o tempo médio de acesso a disco da máquina (cerca de 5 milisegundos). Na execução da consulta, eliminamos o tempo gasto com acessos a disco ficando apenas com o tempo de computação da consulta (tempo de CPU). Este tempo de acessos a disco estava sendo influenciado pelo cache do sistema operacional. Desta forma o tempo real de acessos a disco foi calculado medindo-se o número de acessos a disco necessários e

multiplicando-se este valor pelo tempo médio de acessos a disco calculado previamente. O tempo total de consulta é então a soma do tempo de CPU com este tempo real de acessos a disco calculado.

5.3. Conjuntos de Dados e Consultas

Para nossos testes de avaliação de desempenho utilizamos conjuntos de dados gerados pseudo-aleatoriamente através do gerador de dados espaçotemporais chamado GSTD (do inglês *Generate Spatio Temporal Data*) proposto em THEODORIDIS et al. (1999). Os objetos destes conjuntos de dados são todos retangulares e fizemos pequenas alterações no programa para escalar os retângulos de forma que estivessem contidos numa janela de (0, 0) a (1.000, 1.000) por simplicidade.

Cada conjunto de dados possui um número inicial de inserções e depois movimentos. Procuramos fazer com que o número de movimentos seja mais ou menos igual ao número inicial de inserções. Desta forma utilizamos conjuntos de dados com 100.000 objetos inseridos inicialmente, o que gera 200.000 inserções aproximadamente e 100.000 remoções, num total de 300.000 operações. Criamos conjuntos de dados contendo 250, 500, 750 e 1.000 versões. Para cada tipo de conjunto de dados criamos dois pares de conjuntos.

Para cada conjunto de dados efetuamos uma bateria de 50 consultas de seleção por instante de tempo e por intervalo de tempo e também 50 consultas de junção espaçotemporal por instante de tempo e por intervalo de tempo. Para as análises de ocupação de espaço, de tempo de construção dos índices e de desempenho dos algoritmos de consulta de seleção foi utilizado somente um dos conjuntos que formam um par. Logicamente, a consulta de junção espaçotemporal não foi executada para os conjuntos de dados da HR-Tree, por não estar implementada como foi dito na Seção 5.1. A janela de seleção espacial bem como o intervalo de tempo (para consultas por intervalo de tempo) ou o instante de tempo (para consultas por instante de tempo) foram escolhidos de forma aleatória.

5.4. Tamanho da Página

Para iniciarmos nossa análise experimental procuramos decidir um tamanho de página ideal que dê melhores rendimentos aos algoritmos implementados. Efetuamos uma série de consultas aleatórias em um dos conjuntos de dados (com 100.000 objetos inicialmente inseridos e 500 versões) testando tamanhos de página de 1024, 2048, 4096 e 8192 bytes.

Em todas as nossas implementações dos índices espaçotemporais o tamanho do nó é determinado pelo tamanho de página escolhido. Isto trará impacto nos tamanhos dos índices gerados, no tempo de construção dos índices e nos tempos das consultas de seleção e de junção espaçotemporal. Nesta análise inicial não estamos interessados em comparar os métodos uns com os outros e, por isso, os resultados foram separados por método de acesso na Tabela 5, Tabela 6, e Tabela 7.

Podemos reparar nestas tabelas que o tamanho dos índices gerados não sofre muita alteração com o aumento do tamanho da página para a TR-Tree e para a 2+3D R-Tree. Já para a HR-Tree, quanto maior o tamanho da página, maior o tamanho dos índices gerados, chegando a passar de 1 GBytes para páginas de 8192 bytes. Na HR-Tree muitos nós são duplicados e quanto maior o tamanho da página, maior a quantidade de duplicação em bytes.

Tamanho da Página (bytes)	Tamanho do Índice (MBytes)	Tempo de Construção do Índice (s)	Tempo total das consultas por instante de tempo (s)	Tempo total das consultas por intervalo de tempo (s)	Tempo total das junções por instante de tempo (s)	Tempo total das junções por intervalo de tempo (s)
1024	7,98	493,242	334,12	348,73	1.172,43	1.933,32
2048	7,88	690,978	209,62	228,03	714,51	1.810,10
4096	7,80	1.181,38	123,81	136,01	618,33	1.679,91
8192	7,81	2.193,07	77,89	86,53	801,59	2.320,09

Tabela 5. Parâmetros de verificação do tamanho de página ótimo – 2+3D R-Tree

O tempo de construção dos índices é um fator importante e não pode ser retirado de nenhuma análise sobre indexação em bancos de dados. Podemos reparar que o tempo de construção dos índices possui um comportamento inverso ao tamanho, ou seja,

quanto maior o tamanho das páginas em disco, maior o tempo de construção dos índices para todos os métodos de acesso em análise.

Tamanho da Página (bytes)	Tamanho do Índice (MBytes)	Tempo de Construção do Índice (s)	Tempo total das consultas por instante de tempo (s)	Tempo total das consultas por intervalo de tempo (s)	Tempo total das junções por instante de tempo (s)	Tempo total das junções por intervalo de tempo (s)
1024	18,39	126,19	262,35	521,39	2.076,15	6.906,65
2048	17,98	182,25	145,70	294,33	996,05	3.624,68
4096	16,81	204,18	68,54	134,48	240,07	859,16
8192	16,31	302,75	39,43	72,06	175,97	501,42

Tabela 6. Parâmetros de verificação do tamanho de página ótimo – TR-Tree

O tempo total das consultas de seleção por instante de tempo e por intervalo de tempo têm comportamento semelhante. Ambas as consultas possuem ganho de desempenho com o aumento do tamanho da página para todos os três métodos de acesso em questão.

Tamanho da Página (bytes)	Tamanho do Índice (MBytes)	Tempo de Construção do Índice (s)	Tempo total das consultas por instante de tempo (s)	Tempo total das consultas por intervalo de tempo (s)	Tempo total das junções por instante de tempo (s)	Tempo total das junções por intervalo de tempo (s)
1024	334,46	226,59	110,86	20.922,73	-	-
2048	530,40	365,04	56,17	10.256,58	-	-
4096	833,08	609,08	29,75	5.386,17	-	-
8192	1.145,82	913,88	16,63	3.171,02	-	-

Tabela 7. Parâmetros de verificação do tamanho de página ótimo – HR-Tree

Como a consulta de junção espaçotemporal não foi implementada para a HR-Tree não temos resultados para este método de acesso. Os resultados da consulta de junção espaçotemporal tanto para instante de tempo quanto para intervalo de tempo na TR-Tree possuem comportamento semelhante ao das consultas de seleção, ou seja, quanto maior o tamanho da página utilizado, melhor o desempenho das consultas de junção. Já a 2+3D R-Tree possui um comportamento um tanto quanto diferente. Até páginas de 4096 bytes o tamanho da página melhora o desempenho das consultas de junção, mas para o tamanho de 8192 bytes as consultas de junção já possuem um desempenho pior que as mesmas em um índice com páginas de 4096 bytes. Em todos

estes testes, o número de acessos a disco (tempo de leitura no disco) fica sempre menor conforme o aumento do tamanho de página, mas o tempo de CPU aumenta, como era de se esperar. O que acontece neste caso é que o tempo de CPU começa a ser dominante e a consulta de junção espaçotemporal para a 2+3D R-Tree com páginas de 8192 passa a ser limitada pelo tempo de CPU e não mais pelo tempo de leitura no disco.

Compilando todos estes resultados e comentários resolvemos utilizar em nossos testes páginas de 4096 bytes. O tamanho dos índices não foi deveria ser um fator importante de escolha, mas como a HR-Tree com 8192 bytes de tamanho de página os índices possuíam mais de 1 GBytes, também utilizamos o este fator na escolha do tamanho da página. O tempo de construção dos índices é um fator importante que nos fez escolher por um valor de tamanho de página intermediário, uma vez que este fator possui um comportamento inversamente proporcional ao comportamento do tempo das consultas. O tempo de consultas de seleção pede um tamanho de página máximo. O tempo de junção espaçotemporal, por ser de uma ordem de grandeza maior que o tempo de consultas de seleção, também nos fez escolher o tamanho de página de 4096, devido ao comportamento da 2+3D R-Tree.

5.5. Tamanho dos Índices

O tamanho dos índices é um indicador fraco para avaliação de estruturas de indexação, uma vez que o preço do espaço de armazenamento é cada vez mais baixo. É mais importante a análise de desempenho dos índices para alterações e consultas do que o tamanho que o mesmo ocupa no disco. Mesmo assim resolvemos inserir neste capítulo uma seção contendo uma análise de tamanho dos índices. O gráfico da Figura 20 mostra o tamanho dos índices gerados por conjunto de dados.

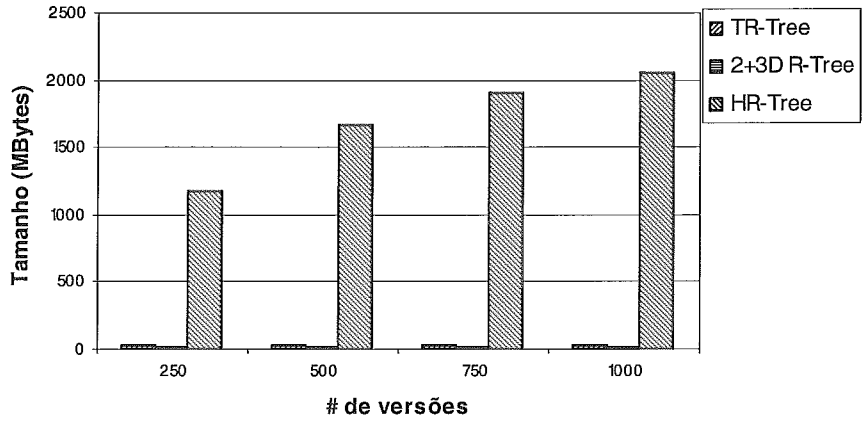


Figura 20. Gráfico dos tamanhos dos indices no disco

Podemos obervar que a HR-Tree possui uma utilização de espaço muito ruim. Embora este parâmetro não seja crucial na escolha da utilização de um índice, a HR-Tree ocupa quase 1 GByte de espaço de armazenamento.

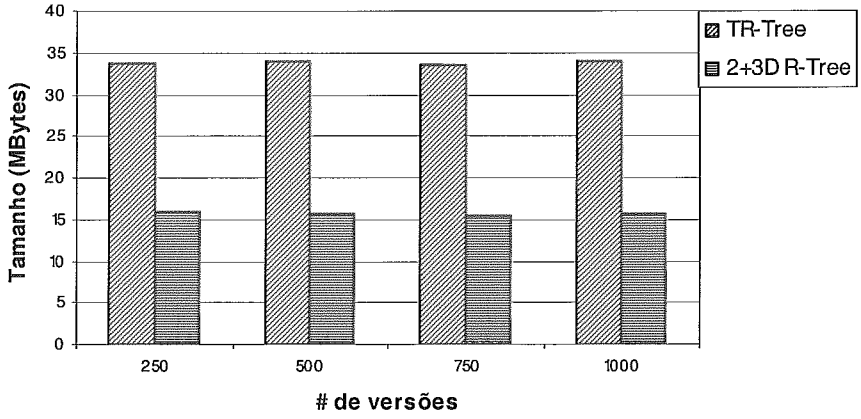


Figura 21. Gráfico dos tamanhos dos indices no disco sem a HR-Tree

Como os valores de ocupação de espaço em disco da HR-Tree encontram-se ordens de grandeza acima dos valores dos outros métodos de acesso, resolvemos mostrar um novo gráfico eliminando a HR-Tree (Figura 21). Podemos perceber neste novo gráfico que a 2+3D R-Tree possui a melhor utilização de espaço em disco das estruturas testadas, embora nem a 2+3D R-Tree nem a TR-Tree sofram com o aumento do número de versões existentes no conjunto de dados.

5.6. Tempo de Geração dos Índices

O tempo de geração dos índices é um bom indicador de desempenho para estruturas de indexação em bancos de dados. O tempo de geração dos índices dividido pelo número total de operações de inserção e remoção efetuadas dá o tempo médio de um operação para a estrutura. A Figura 22 mostra o tempo de geração dos índices para cada conjunto de dados.

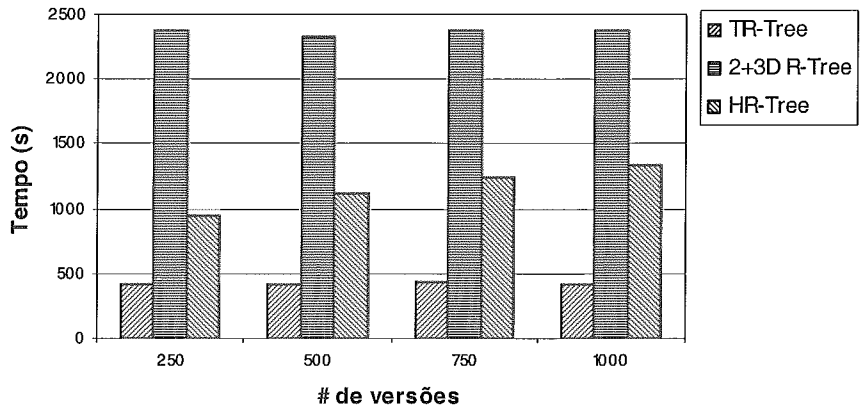


Figura 22. *Gráfico dos tempos de geração dos índices*

Percebe-se neste gráfico que a TR-Tree possui os melhores tempos de geração dos índices, ou seja, o melhor tempo médio de uma operação de inserção ou remoção. Podemos perceber também neste gráfico que o tempo médio de uma operação em nenhuma das estruturas sofre com o aumento do número de versões na mesma.

5.7. Consultas por Instante de Tempo

Nesta Seção iremos fazer uma análise das consultas por instante de tempo. A consulta de seleção por instante de tempo retorna todos os MBRs que interceptavam uma janela R em um instante de tempo t. Como exposto na Seção 5.3 a janela R bem como o instante de tempo t foram escolhidos aleatoriamente. O gráfico da Figura 23 mostra o resultado do somatório de todas as consultas efetuadas por instante de tempo para cada conjunto de dados.

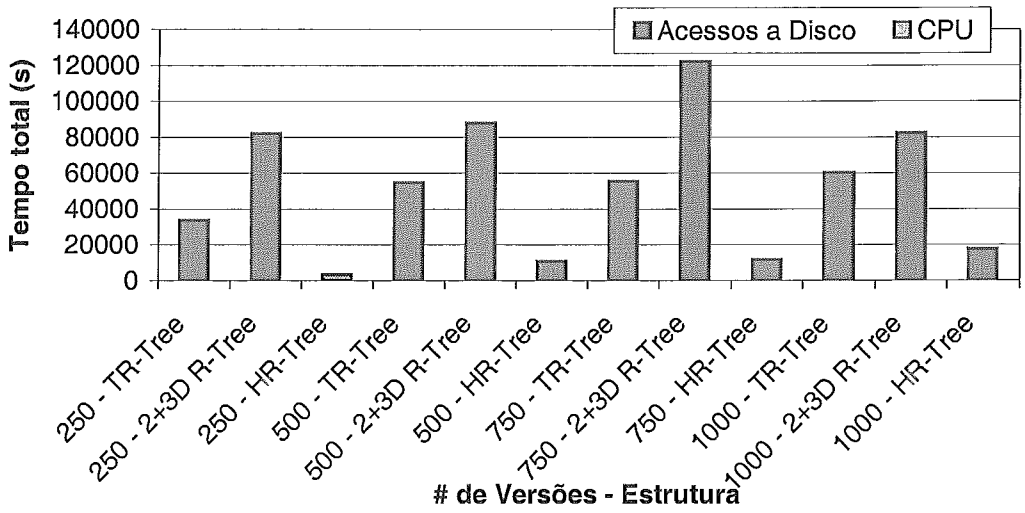


Figura 23. Gráfico de consultas por instante de tempo

Podemos observar nesta figura que a HR-Tree é a estrutura que mostrou os melhores resultados para consultas por instante de tempo. Realmente esperávamos este resultado. Já é apontado na literatura que a HR-Tree é o método de acesso com melhor desempenho para consultas por instante de tempo. A TR-Tree possui desempenho sempre superior à 2+3D R-Tree para estes tipos de consulta, como também era de se esperar. Um outro ponto também importante a ser tocado neste gráfico é que o desempenho das consultas é dado em quase sua totalidade por tempo de acessos a disco, ficando mascarado o tempo de CPU.

5.8. Consultas por Intervalo de Tempo

Nesta Seção iremos fazer uma análise das consultas por intervalo de tempo. A consulta de seleção por intervalo de tempo retorna todos os MBRs que interceptavam uma janela R em um intervalo de tempo $[t_1,t_2)$. Como exposto na Seção 5.3 a janela R bem como o intervalo de tempo $[t_1,t_2)$ foram escolhidos aleatoriamente. O gráfico da Figura 24 mostra o resultado do somatório de todas as consultas efetuadas por intervalo de tempo para cada conjunto de dados. Repare que os resultados da HR-Tree são tão ruins que a escala do gráfico mascara os resultados da TR-Tree e da 2+3D R-Tree, assim como aconteceu no gráfico de ocupação de espaço em disco. Por isso resolvemos mostrar este gráfico sem a presença da HR-Tree. Isto pode ser visto no gráfico da Figura 25.

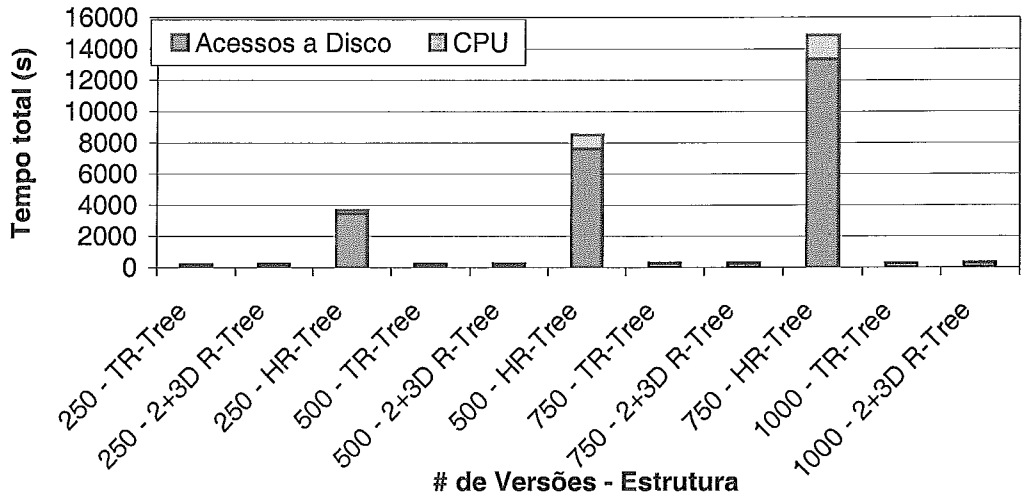


Figura 24. Gráfico de consultas por intervalo de tempo

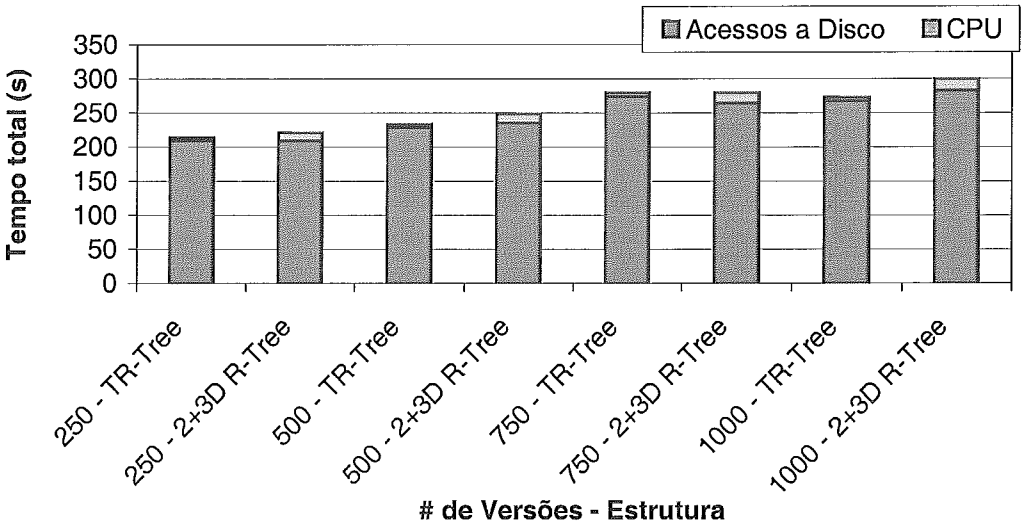


Figura 25. Gráfico de consultas por intervalo de tempo sem a HR-Tree

Podemos observar neste gráfico que o comportamento das duas estruturas em questão (TR-Tree e 2+3D R-Tree) possuem um desempenho muito semelhante, onde a TR-Tree é ligeiramente melhor que a 2+3D R-Tree. Este é um bom resultado observado pois conseguimos com uma estrutura de dados parcialmente persistente (TR-Tree) um desempenho semelhante em consultas por intervalo de tempo a uma estrutura que não duplica entradas (2+3D R-Tree).

Novamente podemos observar que o tempo de acessos a disco é o determinante no tempo total das consultas por intervalo de tempo, mas agora já se consegue perceber

a influência do tempo de CPU, principalmente para a HR-Tree e para a 2+3D R-Tree que possuem muito mais entradas por nó que a TR-Tree.

5.9. Junção Espaçotemporal por Instante de Tempo

Nesta seção iremos analisar o comportamento da TR-Tree e da 2+3D R-Tree diante da consulta de junção espaçotemporal por instante de tempo. A consulta de junção espaçotemporal por instante de tempo retorna todos os pares de MBRs que se interceptavam entre si e interceptavam uma janela R em um instante de tempo t. Como exposto na Seção 5.3 a janela R bem como o instante de tempo t foram escolhidos aleatoriamente. A Figura 26 mostra um gráfico com os somatórios dos tempos calculados para todas as junções executadas por conjunto de dados.

Podemos reparar neste gráfico que a TR-Tree possui um desempenho muito melhor que a 2+3D R-Tree para conjuntos de dados com um número elevado de versões. Podemos verificar também a boa escalabilidade do algoritmo de junção espaçotemporal proposto para a TR-Tree. Os valores de tempo de junção são praticamente os mesmos para ambos os conjuntos de dados. Já o algoritmo de junção espaçotemporal proposto para a 2+3D R-Tree não possui tal comportamento.

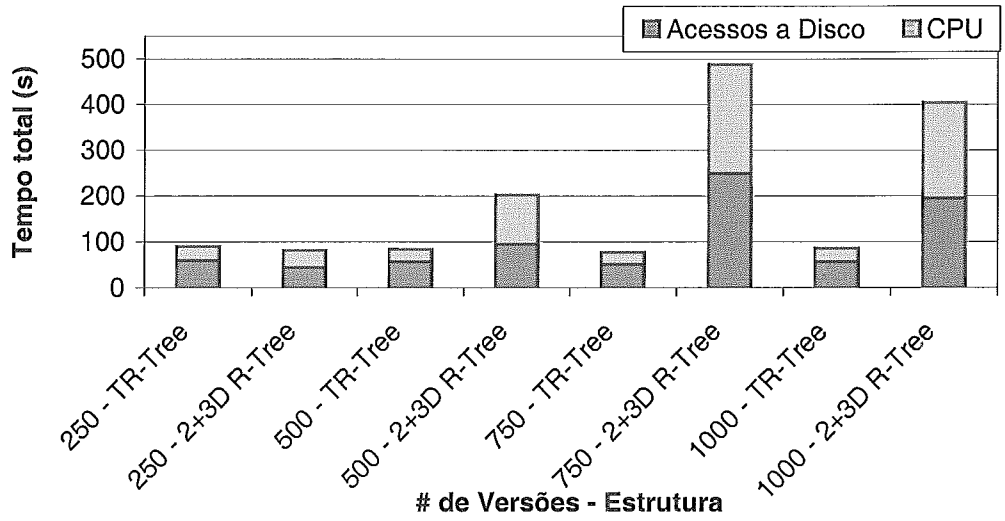


Figura 26. Gráfico de junção espaçotemporal por instante de tempo

O tempo de CPU, para a avaliação das junções, já não é desprezível em relação ao tempo de acessos a disco e, no caso da 2+3D R-Tree, os dois tempos têm praticamente a mesma influência no tempo total de consultas.

5.10. Junção Espaçotemporal por Intervalo de Tempo

Nesta seção iremos analisar o comportamento das duas estruturas (TR-Tree e 2+3D R-Tree) para a consulta de junção espaçotemporal por intervalo de tempo. A consulta de junção espaçotemporal por intervalo de tempo retorna todos os pares de MBRs que se interceptavam entre si e interceptavam uma janela R em um intervalo de tempo $[t_1,t_2)$. Como exposto na Seção 5.3 a janela R bem como o intervalo de tempo $[t_1,t_2)$ foram escolhidos aleatoriamente. A Figura 27 mostra um gráfico com os somatórios dos tempos de acessos a dicso e CPU calculados para todas as junções executadas por conjunto de dados.

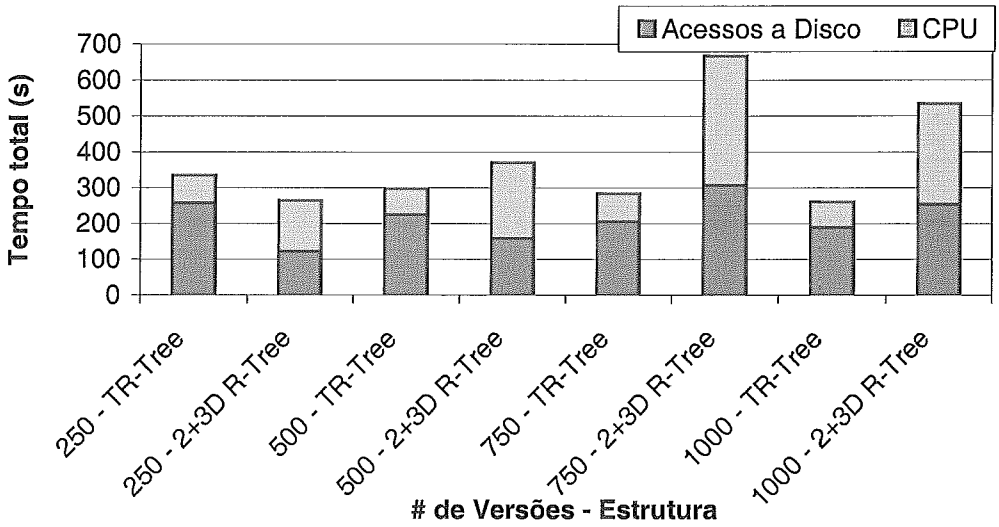


Figura 27. ***Gráfico de junção espaçotemporal por intervalo de tempo***

Podemos verificar novamente a bom comportamento do algoritmo de junção espaçotemporal da TR-Tree diante do aumento do número de versões na estrutura. A 2+3D R-Tree novamente não possui tal comportamento, embora a discrepância entre os tempos de consulta para poucas versões e muitas versões não seja tão grande quanto no caso anterior da junção por instante de tempo.

Assim como na junção por instante de tempo, o tempo de CPU não é desprezível em relação ao tempo de acessos a disco, chegando a ser, em alguns casos, igual ao tempo de acesso a disco para a 2+3D R-Tree.

5.11. Sumário

Fizemos neste capítulo uma análise experimental do método de acesso espaçotemporal proposto nesta dissertação, a TR-Tree. Implementamos todos os seus algoritmos propostos e comparamos seu desempenho com outros dois métodos de acesso existentes na literatura, a 2+3D R-Tree e a HR-Tree.

A TR-Tree possui uma boa ocupação de espaço, perdendo apenas para a 2+3D R-Tree, mas com boa escalabilidade para o aumento do número de versões. A TR-Tree têm os melhores tempos de construção dos índices.

Sabe-se da literatura que a 2+3D R-Tree é um método de acesso muito eficiente para a realização de consultas por intervalo de tempo e que a HR-Tree é um método de acesso muito eficiente para a realização de consultas por instante de tempo. Conseguimos mostrar um bom desempenho em ambos os tipos de consulta. Já nas consultas de junção espaçotemporal tanto por instante de tempo quanto por intervalo de tempo a TR-Tree foi a que mostrou os melhores resultados de desempenho para um número elevado de versões na base de dados.

O próximo capítulo apresenta nossas conclusões e considerações finais, bem como propostas de trabalhos futuros.

6. Conclusões

Nesta dissertação propomos um método de acesso espaçotemporal, a TR-Tree. Propomos também os seus principais algoritmos de inserção, remoção e consultas de seleção e junção espaçotemporal. Foi mostrado o bom desempenho dos algoritmos propostos através de uma avaliação experimental, comparando a TR-Tree a outros métodos de acesso existentes na literatura.

Concluímos que, de todos os novos métodos de acesso espaçotemporais que surgiram na literatura recentemente, a TR-Tree é o mais promissor deles, pois apresenta um bom desempenho em todos os tipos de consulta, o que não acontece para as outras estruturas implementadas.

6.1. Contribuições da Tese

Podemos anotar como contribuições desta tese a implementação de uma estrutura de dados parcialmente persistente, a TR-Tree. A TR-Tree é um método de acesso espaçotemporal eficiente tanto para consultas por instante de tempo como para intervalo de tempo. Um estudo pioneiro em junções espaçotemporais também pode ser creditado como contribuições desta dissertação.

A avaliação da TR-Tree pode ser desmembrada em duas partes: as consultas de seleção e as consultas de junção espaçotemporal. Um artigo (ZIMBRÃO, 2000) foi escrito contendo uma avaliação dos algoritmos da TR-Tree de consultas por seleção e outro contendo uma avaliação dos algoritmos de junção espaçotemporal será apresentado no Simpósio Brasileiro de Bancos de Dados em outubro de 2001 (ZIMBRÃO, 2001).

6.2. Trabalhos Futuros

Como trabalhos futuros, podemos citar a realização mais abrangente da análise experimental, utilizando conjuntos de dados maiores e, se possível, dados de aplicações reais. Consideramos também a possibilidade de paralelização dos algoritmos da TR-Tree.

Algumas novas estruturas têm surgido e principalmente propondo o armazenamento de objetos em movimento. Um estudo da viabilidade do uso da TR-Tree para o armazenamento destes novos tipos de dados pode também ser visto como um trabalho futuro desta dissertação.

Bibliografia

ABRAHAM, T., RODDICK, J. F., 1999, "Survey of Spatio-Temporal Databases", *Geoinformatica*, v. 3, n. 1, pp.61-99.

BECKER, B., GSCHWIND, S., OHLER, T., et al., 1996, "An Asymptotically Optimal Multiversion B-tree", *The VLDB Journal*, v. 5, n. 4 (Dec), pp. 264-275.

BECKMANN, N., KRIEGEL, H. P., SCHNEIDER, R., et al., "The R*-tree: An Efficient and Robust Access Method for Points and Rectangles", In: *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data*, pp. 322-331, Atlantic City, NJ, May.

BLIUJUTE, R., JENSEN, C. S., SALTENIS, S., et al., 1998, "R-tree Based Indexing of Now-Relative Bitemporal Data", In: *Proceedings of 24rd International Conference on Very Large Data Bases*, pp. 345-356, New York City, New York, August..

BRINKHOFF, T., KRIEGEL, H. P, SEEGER, B., 1993. "Efficient processing of Spatial Joins Using R-Trees". In: *Proceedings of the 1993 ACM-SIGMOD International Conference*, Washington, DC, USA, May 1993.

BRODAL, G. S., 1996, "Partially Persistent Data Structures of Bounded Degree with Constant Update Time", In: *Nordic Journal of Computing*, v. 3, n. 3, pp. 238-255.

BURROUGH, P., 1986, *Principles of Geographical Information Systems for Land Resource Assessment*, Oxford, Clarendon Press.

BURTON, F. W., HUNTBACH, M. W., KOLLIAS, J., 1985, "Multiple generation text files using overlapping tree structures", *The Computer Journal*, v. 28, n. 4, pp. 414-416.

CLIFFORD, J., DYRESON, C. E., ISAKOWITZ, T., et al., 1997, "On the semantics of "Now" in temporal databases", ACM Transcactions on Database Systems, v. 22, n. 2 (Jun), pp. 171-214.

COMER, D., 1979, "The Ubiquitous B-tree", *Computing Surveys*, v. 11, n. 2 (Jun), pp. 121-137.

GUNADHI, H., SEGEV, A., 1991, "Query Processing Algorithms for Temporal Intersection Join", In: *Proceedings of the Seventh International Conference on Data Engineering*, pp. 336-344, Kobe, Japan, April.

GUTTMAN, A., 1984. "R-Trees: A Dynamic Index Structure for Spatial Searching". In: *Proceedings of the ACM SIGMOD Intl. Conf on Management of Data*, Boston, MA, USA, May 1984.

JENSEN, C. S., DYRESON, C. E., BÖHLEN, M., et al., 1998, "The Consensus Glossary of Temporal Database Concepts - February 1998 Version", In: *Temporal Databases: Research and Practice*, v. 1399, *Lecture Notes in Computer Science*, Springer-Verlag, pp. 367-405.

KAMEL, I., FALOUTSOS, C., 1994, "Hilbert R-tree: An Improved R-tree using Fractals", In: *Proceedings of the 20th International Conference on Very Large Data Bases*, pp. 500-509, Santiago de Chile, Chile, September.

KOLLIOS, G., GUNOPULOS, D., TSOTRAS, V. J., et al., 2001, "Indexing Animated Objects Using Spatiotemporal Access Methods". To appear at IEEE Transactions on Knowledge and Data Engineering, September.

KOLOVSON, C. P., STONEBRAKER M., 1991, "Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data", In: *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pp. 138-147, Denver, Colorado, May.

KUMAR, A., TSOTRAS, V. J., FALOUTSOS, C., "Access Methods for Bitemporal Databases". In: *Recent Advances in Temporal Databases*, Springer Verlag, pp. 235-254.

KUMAR, A., TSOTRAS, V. J., FALOUTSOS, C., 1997, *Designing Access Methods for Bitemporal Databases*, TR3764, Dept. of Comp. Science, University of Maryland.

LANGRAN, G., 1992, *Time in Geographical Information Systems*, London, Taylor & Francis.

MANOLOPOULOS, Y., KAPETANAKIS, G., 1990, "Overlapping B+-trees for Temporal Data", In: *Proceedings of the 5th Jerusalem Conference on Information Technology (JCIT)*, pp. 491-498.

NASCIMENTO, M.A., DUNHAM, M.H. and ELMASRI, R., 1995, "Using Incremental Trees for Space Efficient Indexing of Bitemporal Databases", In: *Proceedings of the Second International Conference on Application of Databases (ADB'95)*, pp. 235-248, Santa Clara, CA., December.

NASCIMENTO, M.A., DUNHAM, M.H. and ELMASRI, R., 1996a, "M-IVTT: A Practical Index for Bitemporal Databases", In: *Proceedings of the 7th International Conference on Database and Expert Systems Applications (DEXA'96)*, pp. 779-790, Zurich, Switzerland, September.

NASCIMENTO, M.A., DUNHAM, M.H. and KOURAMAJIAN, V., 1996b, "A Multiple Tree Mapping-Based Approach for Valid-Time Ranges Indexing", *Journal of the Brazilian Computer Society*, v. 2, n. 3(Apr), pp. 36-46.

NASCIMENTO, M. A., SILVA, J. R. O., 1998, "Towards Historical R-trees". In: *Proceedings of ACM Symposium on Applied Computing (ACM-SAC)*, pp. 235-240, Atlanta, GA USA, March.

OHLER, T. B. G., 1994, Ph.D. dissertation, Swiss Federal Institute of Technology, Zürich, Switzerland.

ORENSTEIN, J. A., 1986, "Spatial Query Processing in an Object-Oriented Database System", In: *Proceedings of the 1986 ACM SIGMOD International Conference on Management of Data*, pp. 326-333, Washington, D.C., May.

PAPADIAS, D., THEODORIDIS, Y., 1997, "Spatial Relations, Minimum Bounding Rectangles, and Spatial Data Structures", *International Journal of Geographic Information Science*, v. 11, n. 2 (Mar), pp. 111-138.

SALTENIS, S., JENSEN, C., LEUTENEGGER, S., et al., 2000, "Indexing the position of continuously moving points", In: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, pp. 331-342, Dallas, Texas, May.

SAMET, H., 1990. *The Design and Analysis of Spatial Data Structure*, 2 ed, NYC, Addison-Wesley Publishing Company, 1990.

SELLIS, T., ROUSSOPOULOS N., FALOUTSOS C., 1987, "The R+-Tree: A Dynamic Index for Multi-Dimensional Objects", In: *Preceedings of the 13th International Conference on Very Large Data Bases*, pp. 507-518, Brighton, England, September.

SNODGRASS, R. T., 1987, "The Temporal Query Language TQuel", *ACM Transactions on Database Systems*, v. 12, n. 2 (Jun), pp. 247-298.

SNODGRASS, R. T., 1992, "Temporal Databases", In: *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*, v. 639, Springer-Verlag, pp. 22-64.

SNODGRASS, R. T., 1995, *The TSQL2 Temporal Query Language*, Boston, Massachusetts, Kluwer Academic Publishers.

THEODORIDIS, Y., SELLIS, T., PAPADOPOULOS, A. N., et al., 1998, "Specifications for Efficient Indexing in Spatiotemporal Databases", In: *Proceedings of the 10th International Conference on Scientific and Statistical Database Management*, pp. 123-132, Capri, Italy, July.

THEODORIDIS, Y., SILVA, J. R. O., NASCIMENTO, M. A., 1999, "On the Generation of Spatiotemporal Datasets". In: *International Symposium on Spatial Databases (SSD)*, pp. 147-164, Hong Kong, China, July.

THEODORIDIS, Y., VAZIRGIANNIS, M., SELLIS, T., "Spatio-Temporal Indexing for Large Multimedia Applications", In: *Proceedings of the 3rd IEEE Conference on Multimedia Computing and Systems (ICMCS)*, pp. 441-448, Hiroshima, Japan.

TZOURAMANIS, T., VASSILAKOPOULOS, M., MANOLOPOULOS, Y., 1998, "Overlapping Linear Quadtrees: A Spatio-Temporal Access Method". In: *Proceedings of the 6th International Symposium on Advances in Geographic Information Systems*, pp. 1-7, Washington DC, USA, November.

TZOURAMANIS, T., VASSILAKOPOULOS, M., MANOLOPOULOS, Y., 2000, "Multiversion Linear Quadtree for Spatio-Temporal Data". In: *Advances in Databases and Information Systems (ADBIS) - Database Systems for Advanced Applications (DASFAA)*, pp. 215-228, Prague, Czech Republic, September.

XU, X, HAN, J., LU, W., 1990, "RT-tree: An Improved R-tree Index Structure for Spatiotemporal Databases". In: *Proceedings of the 4th International Symposium on Spatial Data Handling (SDH)*, pp. 1040-1049.

ZHANG, D., TSOTRAS, V., SEEGER, B., 2000, *A Comparison of Indexed Temporal Joins*, Time Center Technical Report TR-50.

ZIMBRÃO, G., ALMEIDA, V. T., SOUZA, J. M., 2000, "Efficient Processing of Spatiotemporal Queries in Temporal Geographical Information Systems". In: *Preceedings of the 6th International Conference on Information Systems, Analysis and Synthesis ISAS*, Orlando, Florida USA, August.

ZIMBRÃO, G., ALMEIDA, V. T., SOUZA, J. M., *The Temporal R-Tree*. Technical Report ES492/99, COPPE/Federal University of Rio de Janeiro, March.

ZIMBRÃO, G., ALMEIDA, V. T., SOUZA, J. M., 2001, "Processamento Eficiente de Junções Espaçotemporais", Aceito para Simpósio Brasileiro de Banco de Dados (SBBD'01), Rio de Janeiro, Brasil, Outubro.