Geração e Indexação de Dados Espaço-Temporais

Jefferson Rodrigues de Oliveira e Silva

Dissertação de Mestrado

FICHA CATALOGRÁFICA ELABORADA PELA BIBLIOTECA DO IMECC DA UNICAMP

Silva, Jefferson Rodrigues de Oliveira e

Si38g Geração e indexação de dados espaço-temporais / Jefferson Rodrigues de Oliveira e Silva -- Campinas, [S.P. :s.n.], 1999.

Orientador : Mario A. Nascimento

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

1. Banco de dados. 2. Sistemas de informação geográfica. I. Nascimento, Mario Antonio do. II. Universidade Estadual de Campinas. Instituto de Computação. III. Título.

Geração e Indexação de Dados Espaço-Temporais

Este exemplar corresponde à redação final da Dissertação devidamente corrigida e defendida por Jefferson Rodrigues de Oliveira e Silva e aprovada pela Banca Examinadora.

Campinas, 26 de fevereiro de 1999.

[illegible]

Prof. Dr. Mario A. Nascimento
Universidade Estadual de Campinas
(Orientador)

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

Instituto de Computação
Universidade Estadual de Campinas

Geração e Indexação de Dados Espaço-Temporais

Jefferson Rodrigues de Oliveira e Silva

26 de fevereiro de 1999

TERMO DE APROVAÇÃO

Dissertação defendida e aprovada em 26 de fevereiro de 1999, pela Banca Examinadora composta pelos Professores Doutores:

[illegible]

Prof. Dr. Marcelo Gattass
PUC - Rio

[illegible]

Prof. Dr. Luiz Eduardo Buzato
IC - UNICAMP

[illegible]

Prof. Dr. Mário Antonio Nascimento
IC - UNICAMP

Agradecimentos

Gostaria de expressar meus agradecimentos às pessoas que de uma forma ou de outra contribuiram para a conclusão de mais uma etapa em minha vida:

Minha Mãe, por ter estado sempre ao meu lado e por ter me proporcionado chegar onde estou.

Meu orientador Mario A. Nascimento, pela orientação, ajuda e amizade.

Yannis Theodoridis, for the discussions and lessons learned from our joint work.

Meu Pai e meus Irmãos pelo companherismo e incentivo sempre.

O pessoal da Comunidade Arroba, atual e os que passaram, que me aturaram nesses dois anos.

Os amigos do IC, em especial ao pessoal do grupo de Banco de Dados.

Professores e Funcionários do IC.

FAPESP (processo 97/11205-8) e CAPES pelo apoio financeiro.

Abstract

The goal of the dissertation is the design, implementation and evaluation of an access structure for spatiotemporal data. The dissertation is a collection of four papers written in English, with an introduction and a conclusion written in Portuguese.

The first paper presents a survey of spatial data indices and traditional data persistent indices. In addition, the paper describes a novel structure, the HR-tree, as well as its algorithms to insert, delete, update and search data.

The second paper addresses the development of an algorithm to generate spatiotemporal data, called GSTD (Generate Spatiotemporal Data). The algorithm allows the generation of spatiotemporal data following A few statistical distributions for some user defined parameters, that control, for example, the initial spatial location, the dinamicity of updates (in time) and the spatial data movements.

The third paper presents a comparison of the HR-tree to two other structures. The first one is a 3D spatial structure, based on the R-tree, that treats time as another dimension. In that structure, the initial and end time of the objects have to be known beforehand. The second one is basically a structure that combines two spatial structures, also based on the R-tree: a 2D structure that indexes current objects (i.e., objects with an end time unknown) and a 3D structure that indexes objects alread closed (i.e., objects with initial and end time known).

The fourth and last paper describes an application of the HR-tree in another problem domain, namely bitemporal data indexing.

The overall conclusion of this work is that the HR-tree has the best performance (when compared to the other two structures) to answer spatial queries in a specific point in time and for small time intervals, but the HR-tree is much bigger than the other two structures. However, nowadays space requirements are not as problematic as response time, hence, we believe the HR-tree is a good access structure for spatiotemporal data.

Resumo

O trabalho de dissertação tem como objetivo o desenvolvimento, implementação e teste de uma estrutura de acesso à dados espaço-temporais. A dissertação é uma coleção de quatro artigos escritos em inglês, com uma introdução e uma conclusão escritas em português.

O primeiro artigo faz um levantamento de índices espaciais e índices persistentes de dados tradicionais. Além disso, descreve uma nova estrutura, a HR-tree, bem como algoritmos para inserir, remover, atualizar e consultar dados.

O segundo artigo trata da criação de um algoritmo para geração de dados espaço-temporais, chamado GSTD (Generate Spatiotemporal Data). O algoritmo permite a criação de dados espaço-temporais seguindo algumas distribuições estatísticas para alguns parâmetros definidos pelo usuário, que tratam, por exemplo, da localização espacial inicial, o tempo de duração da instância de um objeto e movimentação dos dados espaciais.

O terceiro artigo apresenta uma comparação da HR-tree com outras duas estruturas. A primeira é uma estrutura espacial 3D, baseada na R-tree, e trata o tempo como outra dimensão. Nessa estrutura, o tempo inicial e final dos objetos têm que ser conhecidos antes de serem inseridos. A segunda é basicamente uma estrutura que combina duas estruturas espaciais, também baseadas na R-tree: uma estrutura 2D que indexa objetos correntes (com tempo final desconhecido), e outra, uma estrutura 3D que indexa objetos já "fechados" (tempos inicial e final conhecidos).

O quarto e último artigo descreve uma aplicação da HR-tree em um outro domínio de problemas, mais precisamente indexação de dados bitemporais.

A conclusão geral do trabalho é que a HR-tree tem o melhor desempenho (quando comparada às duas outras estruturas) em consultas espaciais em um ponto específico no tempo e em intervalos pequenos de tempo, mas a HR-tree é muito maior que as outras duas. No entanto, atualmente espaço não é um problema tão severo como o tempo de resposta, desse modo acreditamos que a HR-tree é uma boa estrutura de acesso à dados espaço-temporais.

Sumário

Lista de Tabelas

Lista de Figuras

Capítulo 1

Introdução

Atualmente, uma grande variedade de aplicações faz uso de dados espaciais, tais como software de Projeto Auxiliado por Computador (CAD) e Sistemas de Informações Geográficas (SIG). Algumas aplicações precisam manter informações passadas sobre objetos, de modo a permitir consultas sobre a evolução no tempo de objetos espaciais, assim como sobre estados passados do banco de dados.

Alguns exemplos de aplicações reais que necessitam controlar dados espaço-temporais incluem armazenamento e manipulação de trajetórias de navios e aviões, previsões de tempo e monitorização de tempestades e incêndios. Além disso, no campo da agricultura, produtores e agências governamentais precisam manter informações de colheitas ao longo do tempo, para tomada de decisões tais como onde usar pesticidas e/ou prover financiamento à fazendeiros. Tais aplicações necessitam de estruturas de acesso eficiente à dados espaço-temporais.

Diferentemente de estruturas de acesso à dados espaciais e de estruturas de acesso à dados temporais, que são bem exploradas na literatura, estruturas de acesso à dados espaço-temporais são pouco exploradas. Nós temos conhecimento de quatro estruturas que consideram tanto as informações espaciais quanto as temporais de objetos. MR-trees e RT-trees [50], 3D R-trees [43] e mais recentemente Overlapping Linear Quadtrees [47]. Somente a 3D R-tree foi implementada e testada, para um domínio de problema específico, indexação em ambientes multimídia. Não temos conhecimento de testes de desempenho das outras estruturas.

O objetivo principal da dissertação foi o projeto de uma estrutura de acesso à dados espaço-temporais, chamada HR-tree (Historical R-tree). A HR-tree é baseada principalmente em dois estudos: o primeiro é a proposta de uma estrutura de acesso à dados espaciais (R-tree [15]) e o segundo uma proposta de uma estrutura persistente de acesso à dados tradicionais (Overlapping B^+-tree [25]). Uma estrutura é dita persistente quando uma atualização cria uma nova versão da estrutura enquanto versões antigas são mantidas e podem ser acessadas [21]. Do mesmo modo que a Overlapping B^+-tree, a HR-tree leva em consideração apenas o tempo de transação de um objeto.

Em banco de dados temporais, o tempo pode ser classificado de duas formas: tempo de validade e tempo de transação. Tempo de validade é o intervalo de tempo no qual um fato ocorreu na realidade, ou seja, é o tempo que ocorreu no mundo real, independente do seu armazenamento no banco de dados. Em tempo de validade, um fato pode ocorrer num futuro conhecido. Tempo de transação é o intervalo de tempo no qual um fato foi armazenado no banco de dados, ou seja, do momento em que foi inserido ao momento em que foi removido [17]. Se um banco de dados suporta as duas dimensões de tempo, é chamado de bitemporal.

Além de propor a HR-tree, foi parte do trabalho de dissertação a implementação e teste da estrutura, através de comparações com outras duas estruturas. A primeira (3D R-tree) [43] indexa objetos 3D, sendo o tempo a terceira dimensão. A segunda (2+3 R-tree) utiliza duas estruturas espaciais, uma 2D e outra 3D. A 2D mantém objetos espaciais correntes, ou seja, seu tempo de transação final não é conhecido. Uma vez que esses objetos são atualizados (fechados, ou seja, seu tempo de transação final é conhecido) são removidos da 2D e inseridos na 3D, como objetos tri-dimensionais (sendo o tempo a terceira dimensão). A estrutura 2+3 R-tree é baseada na estrutura proposta em [20]. As duas estruturas são melhores descritas no capítulo 4.

Para a validação e testes comparativos da HR-tree, era necessário a utilização de dados espaço-temporais. A geração de dados espaciais sintéticos tem sido investigada na literatura [14]. Porém, não conhecemos trabalhos relacionados à geração de dados espaço-temporais. Deste modo, um trabalho desenvolvido na tese foi o estudo da geração sintética de dados espaço-temporais. Um algoritmo para geração sintética de tais dados foi criado, e permite a geração de conjuntos de dados seguindo algumas distribuições estatísticas para alguns parâmetros definidos pelo usuário. Basicamente, os parâmetros tratam da localização espacial inicial, o tempo de duração da instância de um objeto e movimentação dos dados espaciais.

Após termos desenvolvido uma primeira versão do algoritmo, foram realizados testes comparativos entre a HR-tree, 3D R-tree e a 2+3 R-tree. Como é comum na área de indexação, os testes avaliaram três fatores:

- tamanho do arquivo de índice gerado,
- número de páginas de disco acessadas na construção do índice,
- número de páginas de disco acessadas em consultas.

Foi parte do trabalho de dissertação ainda a aplicação da HR-tree em outro domínio de problemas, a indexação de dados temporais. Os trabalhos mais recentes nessa área [5, 4, 21] utilizam estruturas espaciais (R*-tree [1]) na criação de índices bitemporais. De modo semelhante, pensou-se na utilização da HR-tree para indexar dados temporais.

Esta dissertação de tese está organizada como uma coletânea de 4 artigos (capítulos 2, 3, 4, e 5), apresentados na sua forma original. O capítulo 2 apresenta um artigo que descreve a estrutura de índice proposta, a HR-tree. O capítulo 3 apresenta o estudo para geração sintética de dados espaço-temporais e o algoritmo desenvolvido. No capítulo 4 são apresentados testes comparando a HR-tree, a 3D R-tree e a 2+3 R-tree. O capítulo 5 apresenta a aplicação da HR-tree na indexação de dados bitemporais. Finalmente o capítulo 6 apresenta a conclusão da dissertação e possíveis extensões à mesma.

Capítulo 2

Uma Estrutura de Acesso à Dados Espaço-Temporais: HR-tree

Prólogo

No primeiro artigo, é feita uma breve revisão bibliográfica de indexação de dados espaciais, indexação persistente de dados tradicionais e é apresentada uma nova estrutura de indexação de dados espaço-temporais, a HR-tree.

A HR-tree utiliza o conceito de *overlapping trees* [25], ou seja, dado duas árvores sendo a segunda a evolução da primeira (isto é, a segunda é uma versão da primeira baseada nas informações desta), a segunda é representada incrementalmente. Deste modo, apenas os ramos modificados são armazenados, e os não modificados são compartilhados pela nova versão.

R-trees [15] são as estruturas espaciais usadas como base de nosso trabalho. Portanto, a HR-tree utiliza várias R-trees lógicas para manter as informações antigas e atuais de objetos espaciais.

O artigo apresenta a estrutura da HR-tree, os algoritmos básicos de inserção, remoção e atualização de objetos, além dos algoritmos para consultas espaço-temporais.

É preciso salientar que até a conclusão do artigo não tínhamos conhecimento de outras estruturas de acesso a dados espaço-temporais. Desse modo, não é apresentada uma revisão bibliográfica sobre tais estruturas. Essa revisão é apresentada nos artigos dos capítulos 3 e 4.

O artigo foi publicado nos anais do *1998 ACM Symposium on Applied Computing* [27].

TOWARDS HISTORICAL R-TREES

Mario A. Nascimento
CNPTIA – EMBRAPA
P.O. Box 6041
13083-970 Campinas SP BRAZIL
mario@cnptia.embrapa.br

Jefferson R. O. Silva
IC – UNICAMP
P.O. Box 6176
13083-970 Campinas SP BRAZIL
972147@dcc.unicamp.br

Abstract

R-trees are the "de facto" reference structures for indexing spatial data, namely the minumum bounding rectangles (MBRs) of spatial objects. However, R-trees, as currently known, do not support the evolution of such MBRs. Whenever an MBR evolves, its new version replaces the old one, which is therefore lost. Thus, an R-tree always shows the current state of the data set, not allowing the user to query the spatial database with respect to past states. In this paper we extend the R-tree in such a way that old states are preserved, allowing the user to query them. The proposed approach does not duplicate nodes which were not modified, thus saving considerable storage space. On the other hand, the query processing time does not depend on the number of past states stored.

2.1 Introduction

Managing multidimensional data is needed in many application domains, e.g., spatial databases and geographical information systems [33]. In particular, indexing spatial data is of foremost importance in many application domains, and indeed such an issue has been quite well researched. Samet [33] and Gaede and Günther [11] present excellent surveys on the area. However, it is hardly argueable that an structure has been more cited and used as a reference than the R-tree [15]. The R$^+$-tree [34] and the R*-tree [1] are well known R-tree derivatives, where the R*-tree has been shown to be quite efficient. Recently, the Hilbert R-tree [18] and the STR-tree [23] have been shown to have better "packing" capabilities. This is specially useful for sets of data which are not very dynamic in nature. With the exception of the R$^+$-tree, all have the same basic structure. K-dimensional spatial objects are modeled by their Minimum Bounding Rectangles (MBRs) – we assume, without loss of generality, that $K = 2$. Subsets of the indexed MBRs are organized into overlapping subspaces, using a tree hierarchy. They all differ in the way the tree nodes are split (when overflown) and/or MBRs are assigned to subspaces (i.e., nodes in tree).

In this stage of our research we assume an R-tree has already been built by using algorithms from any of the R-tree derivatives, but the R$^+$-tree. Regardless of how it was built, we refer to such an structure as an R-tree, and we assume it obeys the following conditions [15] (where a hyper-rectangle is an MBR):

- Every leaf (non-leaf) node contain between m and M index records (children) unless it is the root;
- For each index record (I, Tid) in a leaf node, I is the smallest hyper-rectangle that spatially contains the object represented by the indicated tuple Tid;
- For each entry (I, Cid) in a non-leaf node, I is the smallest hyper-rectangle that spatially contains the hyper-rectangles in the child node Cid;
- The root node has at least two children unless it is a leaf;
- All leaves appear on the same level.

For simplicity, and proof-of-concept, we base our algorithms for insertion, deletion and updating on those by Gutmann [15]. MBRs can model objects which vary with time, one trivial example is a farm which can be expanded or shrunk by selling or buying land. Similarly new objects can begin or cease to exist. An R-tree indexes only the current MBR for each object. Should any object evolve and have its MBR changed, the R-tree must delete the old MBR and insert the new one (the one corresponding to the new instance of the object). From that point on, no query will ever take into account that past instance of that particular

MBR. In other words, R-trees as currently known, allow querying *only* the current state of a spatial database. A trivial way to overcome such shortcoming would be to store all previous states of the R-trees. As we shall see shortly, this is not an acceptable, nor practical, solution.

The problem of indexing non-spatial objects (i.e., regular tuples in a relation) over time has been researched by many researchers in the temporal databases community. A thorough survey can be found in [31]. However, to our knowledge no research has been published regarding the temporalization of the R-tree. That is exactly the kernel of our contribution.

We assume the temporal attribute is transaction time. Transaction time interval is the time an object has been stored in the database [17]. Hence an MBR is considered stored in the spatial database since the time it is input *ad infinitum*, or until the point it is updated or deleted. This special feature prevents one of using the quite simple idea of considering time as another spatial dimension, thus using the $K+1$ dimensional space to index MBRs varying over time. The problem is that the current MBR versions would have one of its "sides" being extended continously (notice that the current point in time is always moving forward). Even if one could somehow manage this variable side, it would imply a large overlap ratio among the R-trees' subspaces (which does affect negatively the R-tree's performance).

Hence, this paper addresses the problem of querying and maintaing current and past states of R-trees. To accomplish that the paper is organized as follows. The next Section presents an overview of the rationale behind our approach, which we call Historical R-trees. Section 3 discusses how the R-tree's algorithms need be changed to allow the realization of the Historical R-trees. We conclude in Section 4, presenting some issues which are being currently investigated and/or have potential for future research.

2.2 Enhancing R-trees with Time

The technique we propose is inspired by an idea first presented by Burton and colleagues [7, 6]. The authors proposed the use of overlapping trees to manage the evolution of text files. Later, the idea was generalized by [25] to manage temporally evolving B^+-trees in general. The basic idea behind those techniques was to keep current and past states of the B^+-trees by maintaing the original tree and replicating, from state to state, only the root and the branches which reflect any changes. The unchanged branches were not replicated, but rather were pointed to by the nodes in the new branch. The approach we propose is an extension of the overlapping approach originally proposed by Manolopoulos e Kapetanakis [25] for the B^+-trees, to the R-trees. We call such an approach Historical R-tree (H-R-tree for short). In this paper we concentrate on managing it, rather than benchmarking it.

Let us illustrate the rationale supporting the H-R-tree with the following example. Consider the initial R-tree in Figure 2.1(a) at time T0. Suppose that at time T1 MBR 3 suffers a modification resulting in MBR 3a. Likewise MBR 8 is modified at time T2 yielding MBR

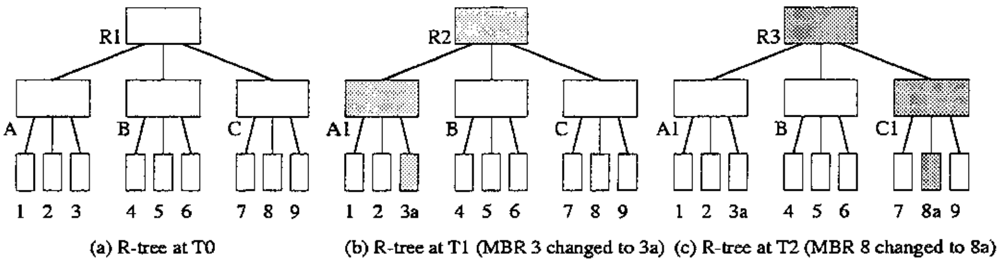


Figure 2.1: An R-tree evolving through time.

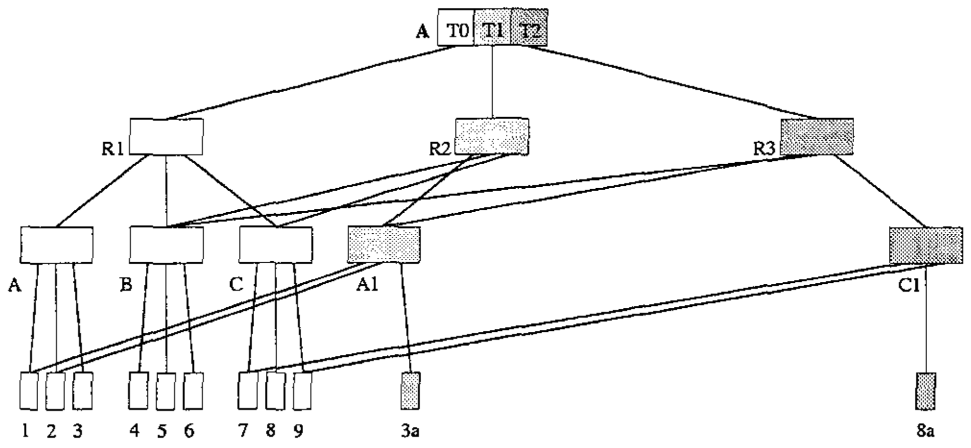
8a. The three states (at T0, T1 and T2) of that particular R-tree are thus those shown in Figures 2.1(a), (b) and (c) respectively. Note that in that particular example the subtree rooted at node B did not change at all, nevertheless it was replicated in all three states. Moreover, the subtree rooted at C (A1) did not change from T0 (T1) to T1 (T2), but the whole subtree was replicated as well. It should be now clear that duplicating the whole tree at each state is rather unpractical.

Let us now see how the same scenario would be handled by the H-R-tree. We assume an array, called A, indexing the time points where updates occurred. The initial R-tree must be kept full and it is pointed to by A[T0]. From T0 to T1 MBR 3 changes, and as such node A changes as well, after all its contents did change. This update propagates upwards until the root node. At the end only the path {R1, A, 3} needs to be updated, resulting the new path {R2, A1, 3a}. Naturally, the R-tree at T1 is composed by the subtrees rooted at A1, B and C. However, those rooted at B and C did not change at all, and thus need not be replicated. Similarly, at time T2 the R-tree rooted at R3 is composed by those subtrees roted at A1, B and C1. Again, from T1 to T2, the subtrees rooted at A1 and B did not go under any modification and as such need not be replicated. The resulting H-R-tree at time T2 is shown in Figure 2.2(a). Simple inspection shows that, even in a trivial example like this, the H-R-tree is much smaller than the set of tree R-trees in Figure 2.1. Figure 2.2(b) shows the logical view of the resulting R-tree at time T2, which is exactly the same one in Figure 2.1(c).

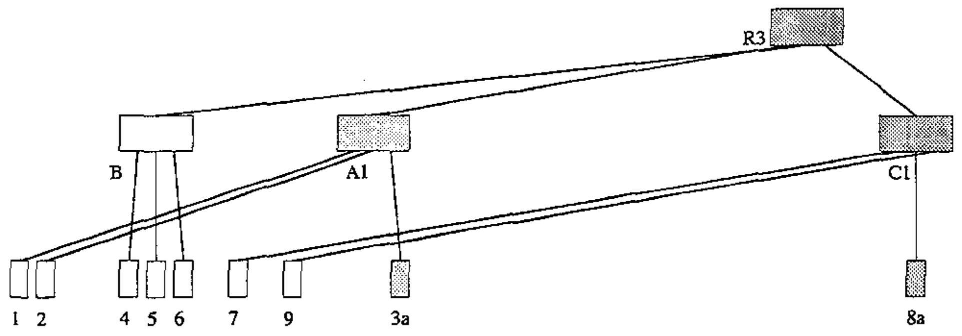
It is important to stress that querying *any* version of the R-tree under the H-R-structure is a matter of obtaining the correct R-tree root. Once this is done, using the array A[.], the logical view is that of a standard R-tree, no complications are therefore added by the approach we use to keep the R-tree's history.

2.3 Historical R-trees

In this section we discuss in detail how to modify the R-tree's algorithms in order to realize the H-R-tree. Given the space constraints we omit certain details, which can be found in



(a) Physical view, semantically equivalent to Figure 1, but using the Historical R-tree.



(b) Logical view of the Historical R-tree state at time T3

Figure 2.2: Physical and logical views of the Historical R-tree.

Guttman's original paper [15].

We draw particular attention to insertion and (logical) deletions of MBRs, modification of MBRs can be accomplished by deleting the old version and inserting the new one. As we argued above, querying any version of the R-tree is straightforward.

The H-R-tree is a structure composed by an array `A` of time values, which in turn point to several logical R-trees (see Figure 2.2). The H-R-tree structure is very similar to the other R-tree derivatives. The difference, besides the existence of array `A` is that each node contains a timestamp `t`, representing the time that one node was created. Any operation is always performed onto the most current R-tree version, i.e., the one pointed to by `A[t]`, and will yield a new version, which is thus timestamped with `t = now`[1]. From now on, we use the notation presented in Table A.1.

Table 2.1: Notation used in the algorithms.

`OR`	a pointer to the H-R-tree
`A`	the H-R-tree's array of time points
`R`	root node of a R-tree
`On`	MBR inserted in the H-R-tree
`Oo`	MBR removed from the H-R-tree
`F`	an entry in a R-tree node
`I`	MBR associated to `F`
`p`	pointer associated to `F`
`N.Nt`	node N's timestamp
`Q`	queue of R-tree nodes
`L,LP,N,NN,NR,P,PP`	pointers to R-tree nodes
`now`	current point in time
`pt`	most recent entry in `A`

2.3.1 Inserting an MBR into the Current R-tree

The `Insert` algorithm inserts a new MBR `On` into the most recent logical version of the R-tree within the H-R-tree `OR`, thus creating a new R-tree version (whose nodes will be timestamped with `now`).

A new branch of the H-R-tree is created, in which `On` is placed. A new logical R-tree rooted at `NR` is created, and it is pointed to by `A[now]`. The algorithm first invokes `CreateBranch` which descends the current R-tree rooted at `R` (which is pointed to by `A[pt]`)

[1]In the temporal database literature (e.g. [38]), `now` is usually regarded as a variable. We, on the other hand, use it only as a shorter name for "the value of the current point in time".

to find the leaf node in which On will be placed. CreateBranch returns a root NR and a leaf node L, where On is to be inserted. A[now] is then updated to NR, obtaining the most recent logical version of the R-tree. (CreateBranch was adapted from the Guttman's ChooseLeaf algorithm.)

```
Algorithm Insert(On, OR)
1. { create a new state in the H-R-tree }
   if A[pt] < now
      then create a new entry in A indexing now;
2. { create a root NR to insert On }
   invoke CreateBranch to create a new logical
     R-tree rooted at NR. The new logical R-tree
     contains a leaf node L in which to place On;
3. { insert On in L }
   if L has room for another entry
      then insert On in L;
      else remove L created by CreateBranch;
           invoke SplitNode to obtain a new L
             and LP containing On and all the
             other entries of L removed;
4. { propagate split upwards }
   if a split was performed
      then Invoke AdjustTree on L, also passing LP;
5. { grow tree taller }
   if node split propagation caused a root split
      then create a new root NR whose children are
             the resulting nodes resulting from the
             root split;
   adjust the entry in A to point to the new root NR;

Algorithm CreateBranch(On, OR)
1. { initialize }
   set N to be the root pointed to by A[pt] in OR;
   if N.Nt < now
      then create a new node L;
           copy all entries of N into L;
           set L.Nt = now;
           set NR = L;
      else set NR = N;
```

```
         set L = N;
2. { leaf check }
   if N is a leaf
      then return NR and L;
3. { choose subtree }
   let F be the entry in N whose rectangle F.I
     needs least enlargement to include On. Break
     ties by choosing the entry with the rectangle
     of smalest area;
4. { create a new node of the new branch and
     descending the tree }
   set LP = L;
   set N to be the node pointed to by F.p;
   if N.Nt < now
      then create a new node L;
           copy all entries of N into L;
           set L.Nt = now;
           adjust the pointer F.p in LP to point
             to L;
      else set L = N;
5. { Loop until a leaf is reached }
   repeat from step 2;
```

The SplitNode and AdjustTree procedures mentioned above are essentially the algorithms defined by Guttman [15]. SplitNode is used when a new object On is inserted into a full node. In this case, all the entries should be divided between two nodes. A small change is needed in the original SplitNode Algorithm though, namely the algorithm must set the timestamp of the newly created nodes. The AdjustTree ascends from a leaf node to the root, adjusting the covering rectangles and propagating splits as necessary. It is interesting to note that the logical view of the H-R-tree is that of a standard R-tree at some point in time, such as now, and as such, node splits are handled as if we were manipulating a standard R-tree.

Figure 2.3 shows an example where from T0 to T1 a new MBR, labelled 7 was inserted. The algorithm decided that it should be inserted in node B, which was already full. Therefore a node split happened, resulting in nodes B1 and B2. As the root R1 was not full, no further splitting is needed. From T1 to T2, no split is needed when MBR 8 is input into node A, thus only a single branch (optimal case) is replicated.

Notice that in the worst case a node split is propagated all the way from leaf to root,

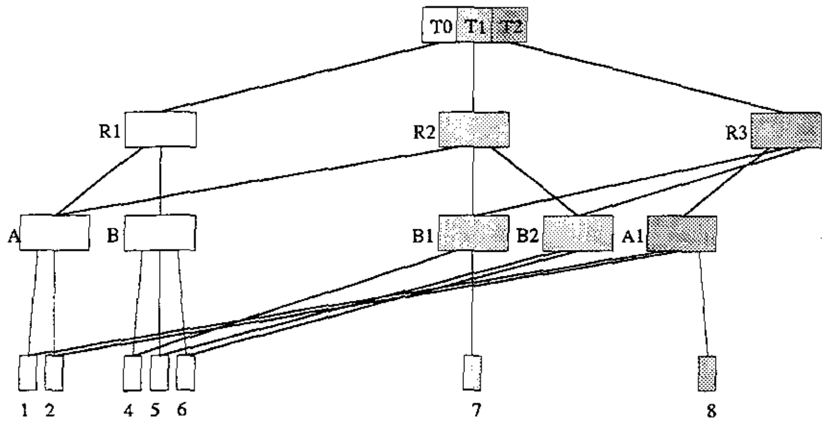


Figure 2.3: Example of how insertion can affect the H-R-tree.

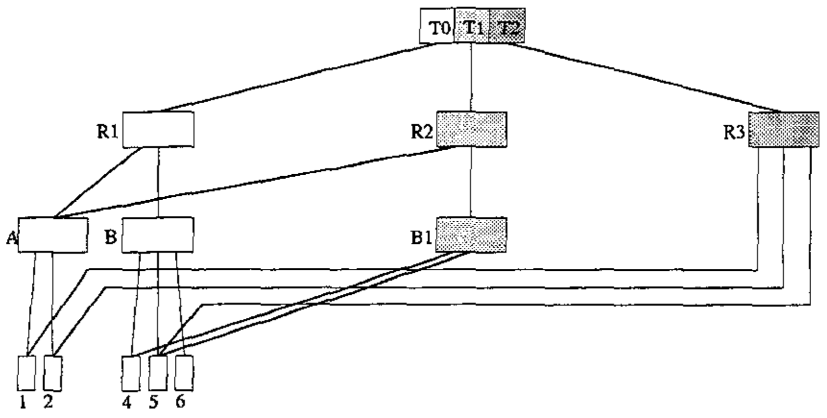


Figure 2.4: Example of how deletion can affect the H-R-tree.

resulting in two (i.e., a constant and low number) branches being replicated. Therefore, considering an R-tree uses $O(n/B)$ space[2], the H-R-tree after u updates uses $O(n/B + u \log_B n)$ space. If all previous R-tree states were kept $O(un/B)$ space would be required.

2.3.2 Deleting (logically) an MBR from the Current R-tree

The `Delete` algorithm removes an MBR `Oo` from an H-R-tree `OR` at time `A[pt]`. A new entry in `A` is created, indexing `now`. A new logical R-tree is created, which reflects the new state of the R-tree without the MBR `Oo`. After `Oo` is removed, `Delete` invokes a slightly modified version of Guttman's CondenseTree algorithm to eliminate the node if it has too few entries and to relocate its entries. These are re-inserted in the R-tree rooted at `A[now]`. The CondenseTree algorithm also propagates node elimination upward as necessary, adjusting covering rectangles. The only two modifications necessary in Guttman's CondenseTree are: (1) in the insertion algorithm used to re-insert the entries removed, the `Insert` algorithm described above must be used instead of the original one, and (2) a node containing entries to be inserted in set Q is only removed if its timestamp is `now`. A node will be removed only if it was duplicated by the `Delete` algorithm, otherwise, it belongs to a previous state of the R-tree, and as such it should not be removed.

```
Algorithm Delete(Oo, OR)
1. { find the leaf node containing Oo}
   set R to be the root pointed to by A[pt];
   set Q, the queue of nodes already traversed,
     to be empty;
   invoke FindLeaf passing R to locate the leaf
     node L containing Oo;
   if L cannot be found
      then stop;
2. { create a new state in the H-R-tree }
   create a new entry in A with time value equal
     to now;
3. { create a new branch without Oo }
   create a new node LP;
   set LP.Nt = now;
   remove the first element of Q and put all of
     its entries in LP;
   set R = LP;
   set A[now] to point to LP;
```

[2]n is the number of indexed MBRs.

```
   while Q is not empty
      create a new node L;
      set L.Nt = now;
      remove the next element N of Q and put all
        of its entries in L;
      adjust the entry in LP pointing to N to point
        to L;
      set LP = L;
   Remove Oo from L;
4. { adjust tree }
   Invoke CondenseTree, passing L;
5. { shorten tree }
   if the root node has only one child after the
   tree has been adjusted
      then make the child the new root R;
   adjust A[now] to point to R;

Algorithm FindLeaf(R)
1. { search subtrees storing the branch that
     contains Oo }
   if R is not a leaf
   then for each entry F in R
           if F.I overlaps Oo
              then if R is not in Q yet
                      then Put R in Q;
              let R be the root of the subtree
                pointed to by F.p;
              invoke FindLeaf passing R;
       until Oo is found or all entries are checked;
       if Oo was not found and R was inserted in Q
          then remove last R inserted into Q;
    else check each entry to see if it matches Oo;
       if Oo is found
          then put R in Q;
       return R;

Algorithm CondenseTree(R)
1. { initialize }
```

```
   set N = L;
   set Q, the set of eliminated nodes, to be empty;
2. { find parent entry }
   if N is the root
      then go to 6;
      else let P be the parent of N;
           let F be N's entry in P;
3. { eliminate under-full node }
   if N has fewer than m entries
      then delete F from P;
           copy the entries of leaf nodes of subtree
             rooted at N to set Q;
           if N.Nt = now
             then remove N;
4. { adjust covering rectangle }
   if N has not been eliminated
      then adjust F.I to tightly contain all entries
        in N;
5. { move up one level in tree }
   set N = P and repeat from 2;
6. { re-insert orphaned entries }
   re-insert all entries of nodes in set Q;
```

Notice that the `CondenseTree` algorithm may re-insert $m-1$ MBRs in the current version of the R-tree. Recall that m is the minimum number of entries occupied in any R-tree node. Therefore, considering the algorithm `Insertion` presented earlied, in the worst case, `Delete` yields $2 \times (m-1)$ branches being replicated. It is worthwhile stressing that this is a constant number.

As an illustration, Figure 2.4 shows an H-R-tree where from time T0 to T1, MBR 6 was deleted. As B1 still remains with its minimum load (assuming $m = 2$), no further modifcation is needed but replicating the branch that lead to MBR 6. From T1 to T2, MBR 4 is deleted and a more interesting situation arises. After deleting MBR 4 B1 falls under its minimum load, and thus all its entries must be re-inserted, freeing that node. Freeing that node leave the root of the current R-tree with only one child, which cannot occur, again per R-tree's definition. A new root (R3) is then created containing the remaining MBRs 1, 2 and 5.

2.3.3 Modifying a Single MBR in the Current R-tree

Modifying an MBR in the current R-tree is rather trivial. All one needs to do is to remove the current version of the MBR `Oo` and insert the new version `On` using the algorithms `Delete` and `Insert` already discussed.

```
Algorithm Modify(Oo, On, OR)
1. { Delete the current MBR version }
   Invoke Delete passing Oo and OR;
2. { Insert the new MBR version }
   Invoke Insert passing On and OR;
```

We have already noted that another `Modify` algorithm could be devised. The main idea is to forcefully enlarge or shrink the MBRs, higher in the tree, that involve the MBR being modified. This would ensure that only one branch would be replicated per MBR modification. The cost of such alternative is yet to be investigated.

2.3.4 Querying the Current R-tree

The `Search` algorithm finds all MBRs in the H-R-tree `OR` that overlap the search windows `S`. The search can be made in some specific past state of the R-tree or in the current state, that is the R-tree pointed to by `A[pt]` The `Search` algorithm first finds the (R-tree) root in the H-R-tree that is pointed to by A[t]. After that, Guttman's original Search algorithm is used to find the MBRs that overlap the search windows `S`.

```
Search(S, t, OR)
1. { find the appropriate root R }
   if t = now
      then set R to be the node pointed to by A[pt];
      else set R to be the node pointed to by A[t];
2. { find the MBRs which overlap S }
   invoke Guttman's original Search algorithm
     passing R;
```

For a discussion on how to query the R-tree we refer the reader to Guttman's original paper [15], as no modifications (at all) to the original algorithm are required.

2.4 Directions for Future Research

We have presented a systematic way to deal with modifications of an R-tree while preserving all of its previous states. While the resulting structure, which we named Historical R-tree

(H-R-tree) saves substantial space when compared to saving all previous R-tree states, it does not degenerate query processing time. Future research is needed in several directions, such as:

- Implementation and benchmarking of the H-R-tree, including the generation of spatial-temporal data;
- Using Z-order [29] along with the original Historical B^+-trees [25] to index spatial-temporal data (and consequently compare the result to the H-R-tree);
- Adapting the idea of Persistent B^+-trees [22] to R-trees (and also compare the result to the H-R-tree);
- Investigate how the proposed approach depends on the initial R-tree configuration (recall one could use any of the R-trees, but the R^+-tree, algorithms to generate the initial R-tree) and
- Determine how concurrency control algorithms are impacted in the proposed approach.

2.5 Acknowledgements

Mario A. Nascimento is also an invited lecturer at Institute of Computing of the State University of Campinas (mario@dcc.unicamp.br). Jefferson R. O. Silva was supported by CAPES.

Capítulo 3

Geração Sintética de Dados Espaço-Temporais

Prólogo

O segundo artigo trata da geração de dados espaço-temporais sintéticos. Vários algoritmos tem sido implementados para geração de dados espaciais estáticos (pontos ou retângulos), seguindo distribuições estatísticas pré-definidas. O mais relevante para nosso trabalho é [14].

Porém, quando a dimensão temporal é adicionada à objetos espaciais, a geração sintética de dados passa a ser um problema mais complexo.

Para que a HR-tree pudesse ser testada e comparada com outras estruturas, era necessário a geração de dados espaço-temporais. Deste modo, parte do trabalho da dissertação foi o estudo da geração desses dados, que resultou na construção de um algoritmo gerador de dados espaço-temporais, denominado GSTD.

O algoritmo proposto trata basicamente de três características dos dados: (1) duração da instância de um objeto; (2) deslocamento de um objeto e (3) mudança no tamanho de um objeto.

O algoritmo permite ao usuário a configuração de alguns poucos parâmetros seguindo algumas distribuições estatísticas. Como pode ser visto no artigo, ajustando os parâmetros de forma adequada, o usuário pode construir vários cenários diferentes. O artigo apresenta ainda alguns exemplos de conjuntos de dados gerados utilizando-se o GSTD.

O artigo foi publicado como relatório técnico CH-99-01 [1] no projeto ChoroChronos, uma rede de pesquisa em sistemas de banco de dados espaço-temporais financiado pela comunidade européia, e foi submetido para publicação no *6th International Symposium on Spatial Databases (SSD).*

[1]Disponível em http://www.dbnet.ntua.gr/~choros/TRs/1999/1/report.ps.gz

On the Generation of Spatiotemporal Datasets

Yannis Theodoridis* Mario A. Nascimento+ Jefferson R. O. Silva+

*Computer Science Division
Dept. of Electrical and Computer Eng.
National Technical University of Athens
Zographou 15773, Athens, HELLAS
theodor@cs.ntua.gr

+Institute of Computing
State University of Campinas
PO Box 6041
13083-970 Campinas SP BRAZIL
{mario, 972147}@dcc.unicamp.br

Abstract

An efficient benchmarking environment for spatiotemporal access methods should at least include modules for storing synthetic and real datasets, collecting and running access structures, and visualizing experimental results. Focusing on the dataset repository module, a wide set of synthetic that would simulate a variety of real life examples in required. Several algorithms have been implemented in the past to generate static spatial (point or rectangular) data, for instance, following a predefined distribution in the workspace. However, by introducing motion, and thus temporal evolution in spatial object definition, generating synthetic data tends to be a complex problem. In this paper, we discuss the parameters to be considered by a generator for such type of data, propose an algorithm, called "Generate_Spatio_Temporal_Data" (GSTD), which generates sets of moving point or rectangle data that follow an extended set of distributions, and visualize some of the results. The GSTD source code and illustrative examples are currently available in the Internet [1]

Keywords: spatiotemporal databases, benchmarking, data generators, indexing, access structures, query performance.

3.1 Introduction

A field of ongoing research in the area of spatial databases and Geographical Information Systems (GIS) involves the accurate modeling of real geographical applications, i.e., applications that involve objects whose position, shape and size change over time. Real world

[1]Mirror sites: http://www.dbnet.ece.ntua.gr/~theodor/GSTD/ and http://www.dcc.unicamp.br/~mario/GSTD/

examples include storage and manipulation of trajectories, fire or hurricane front monitor, simulators (e.g. flight simulators), weather forecast, etc.

Database Management Systems (DBMS) should be extended towards the efficient modeling and support of such applications. Towards this goal, recent research efforts have aimed at:

- modeling and querying time-evolving spatial objects (e.g. [35, 10, 44]),
- designing index structures and access methods (e.g. [27, 47]),
- implementing appropriate architectures and systems (e.g. [48]).

In the recent literature, one can find work on formalization and modeling of spatiotemporal databases and a wide set of definitions about spatiotemporal objects. In the rest of the paper, we adopt the discrete definition for spatiotemporal objects that appears in [40]:

Definition: A *spatiotemporal object*, identified by its *o_id*, is a time-evolving spatial object, i.e., its evolution (or 'history') is represented by a set of instances (*o_id*, s_i, t_i), where s_i is the location of object o at instant t_i (s_i and t_i are called *spacestamp* and *timestamp*, respectively).

According to the above definition, a two-dimensional time-evolving point (region) is represented by a line (solid) in three-dimensional space. Figure 3.1 illustrates two examples: (a) a *moving point* and (b) a *moving region*, according to the terminology proposed in [10]. Although in the rest of the paper, we consider objects of dimensionality $d = 2$, extension to higher dimensions is straightforward [2].

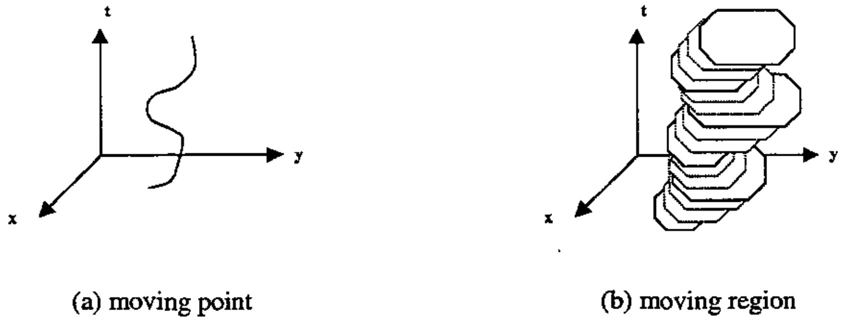


Figure 3.1: Two-dimensional time-evolving spatial objects

One of the tasks that a Spatiotemporal Database Management System (STDBMS) should definitely support includes the efficient indexing and retrieval of spatiotemporal objects. This

[2]Popular examples of spatial datasets with dimensionality $d > 2$ include, among others, virtual reality worlds ($d = 3$) and feature-based image databases (usually $d \geq 256$).

task demands robust indexing techniques and fast access methods for a wide set of possible queries on spatiotemporal data. Either extensions of existing spatial access methods [50, 43, 27, 47] or new 'from-the-scratch' methods could be reasonable candidates. All proposals, however, should be evaluated under extensive experimentation on real and synthetic data. For instance, query processing and/or index building time (either real wall-clock time, or number of disk I/Os), space requirements and combinations thereof are all possible parameters against which one may want to evaluate a given index proposal.

Overall, there is a lack of consistent performance comparison among the proposed approaches, with respect to the space occupied, the construction time, and the response time in order to answer a variety of spatial, temporal, and spatiotemporal queries. Moreover, [52] suggests that *"experiments of indexing techniques should be based on benchmarks such as standard sets of data and queries"*.

Following that, the general architecture of a benchmarking environment for spatiotemporal access methods (STAMs) that is currently under design includes the following:

(a) *a module that generates synthetic data and query sets*, which would cover a variety of real life examples,

(b) *a repository of real datasets* (such as TIGER files for - static - spatial data),

(c) *a collection of access structures* for experimentation purposes,

(d) *a database of experimental results*, and

(e) *a visualization tool* that could be able to visualize datasets and structures, for illustrative purposes.

Our study continues an attempt towards a *specification and classification scheme* for STAMs initiated in [40]. Within the above framework, in this paper we concentrate on module (a) and, in particular:

- discuss parameters that have to be taken into consideration for generating spatiotemporal datasets, and
- propose an algorithm that generates datasets simulating a variety of scenarios with respect to user requirements.

The rest of the paper is organized as follows: In Section 3.2 we discuss the motivation for this study. Section 3.3 discusses the parameters that need to be taken into consideration. An appropriate algorithm is presented in Section 3.4 together with example results and applications. Section 3.5 discusses several issues that arise and surveys related work. Finally, Section 3.6 concludes by also giving directions for future work.

3.2 Motivation

In the literature, several access methods have been proposed for spatial data without, however, taking the time aspect into consideration. Those methods are capable of manipulating geometric objects, such as points, rectangles, or even arbitrary shaped objects (e.g. polygons). An exhaustive survey is found in [12]. On the other hand, temporal access methods have been proposed to index valid and/or transaction time, where space is not considered at all. A large family of access methods has been proposed to support multiversion / temporal data, by keeping track of data evolution over time (e.g. assume a database consisting of medical records, or employees' salaries, or bank transactions, etc.). For a survey on temporal access methods see [31].

To the best of our knowledge, there is a very limited number of proposals that consider both spatial and temporal attributes of objects. In particular, *MR-trees* and *RT-trees* [50], *3D R-trees* [43], and *HR-trees* [27] are based on the R-tree family [15, 1, 18] while *Overlapping Linear Quadtrees* [47] are based on the Quadtree structure [32]. These approaches have the following characteristics:

- 3D R-trees treat time as another dimension using a 'state-of-the-art' spatial indexing method, namely the R-tree,
- MR-trees and HR-trees (Overlapping Linear Quadtrees) embed the concept of overlapping trees [25] into R-trees (Quadtrees) in order to represent successive states of the database, and
- RT-trees couple time intervals with spatial ranges in each node of the tree structure by adopting ideas from TSB trees [24].

The majority of proposed spatiotemporal access structures are based on the R-tree (one exception is [47]), as such we focus on such structures and a short survey of the R-tree based approaches follows.

Assuming *time* to be another dimension is a simple idea, since several tools for handling multidimensional data are already available [12]. The 3D R-tree implemented in [43] considers time as an extra dimension in the original two-dimensional space and transforms two-dimensional rectangles in three-dimensional boxes. Since the particular application considered in [43] (i.e., multimedia objects in an authoring environment) involves Minimum Bounding Rectangles (MBRs) that do not change their location through time, no dead space is introduced by their three-dimensional representation. However, if the above approach were used for moving objects, a lot of empty space would be introduced (Figure 3.2).

The approach followed by the RT-tree [50] only partially solves that problem. Time information is incorporated, by means of time intervals, inside the (two-dimensional) R-tree

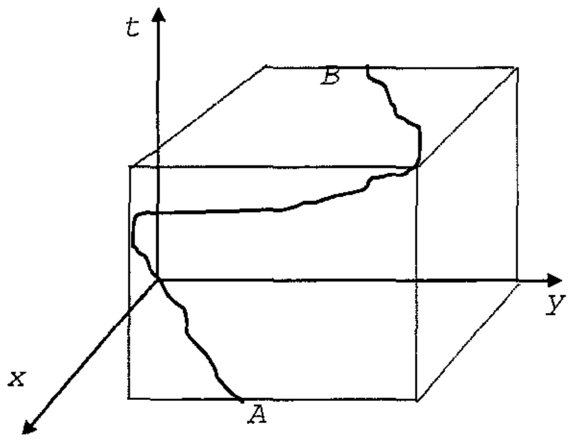


Figure 3.2: The MBR of a moving object occupies a large portion of the data space

structure. Each entry, either in a leaf or a non-leaf RT-tree node, contains entries of the form (*S*, *T*, *P*), where *S* is the spatial information (MBR), *T* is the temporal information (interval), and *P* is a pointer to a subtree or the detailed description of the object. Let $T = (t_i, t_j)$, $i \leq j$, t_j be the current timestamp and t_{j+1} be the consecutive one. If an object does not change its spatial location from t_j to t_{j+1}, then its spatial information *S* remains the same, whereas the temporal information *T* is updated to *T'*, by increasing the interval upper bound, i.e., $T' = (ti, t_{j+1})$. However, as soon as an object changes its spatial location, a new entry with temporal information $T = (t_{j+1}, t_{j+1})$ is created and inserted into the RT-tree. This insertion strategy makes the structure mostly efficient for databases of low mobility; evidently, if we assume that the number of objects that change is large, then many entries are created and the RT-tree grows considerably. An additional criticism is based on the fact that R-tree node construction depends on spatial information *S* while *T* plays a complementary role. Hence the RT-tree is not able to support temporal queries (e.g. *"find all objects that exist in the database within it a given time interval")*.

On the other hand, MR-trees and HR-trees are influenced by the work on overlapping B-trees [25]. Both methods support the following approach: different index instances are created for different transaction timestamps. However, in order to save disk space, common paths are maintained only once, since they are shared among the structures. The collection of structures can be viewed as an *acyclic graph*, rather than a collection of independent tree structures. The concept of overlapping tree structures is simple to understand and implement. Moreover, when the objects that have changed their location in space are relatively few, then this approach is very space efficient. However, if the number of moving objects from one time instant to another is large, this approach degenerates to independent tree structures, since no common paths are likely to be found. Figure 3.3 illustrates an example of overlapping trees for two different time instants t_0 and t_1. The dotted lines represent links to common paths / subpaths.

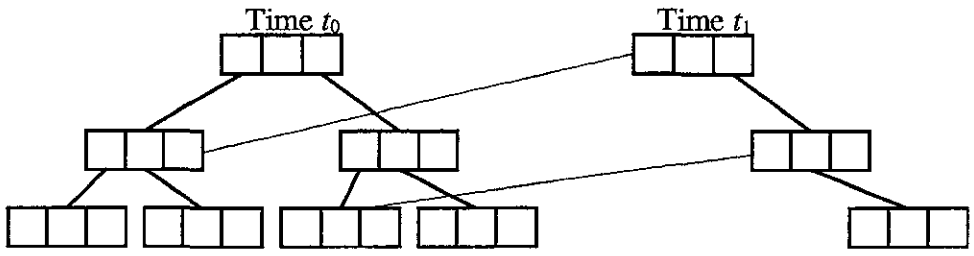


Figure 3.3: Overlapping trees for two different time instants t0 and t1.

Among the aforementioned proposals, the 3D R-tree has been implemented and experimentally tested [43] using synthetic (uniform) datasets. The retrieval cost for several pure temporal, pure spatial and spatiotemporal operators was measured and appropriate guidelines were extracted. Recently, [28] compares the HR-tree with the 3D R-tree and another structure, called 2+3 R-tree, using two R-trees and a rationale similar to the 2R approach presented in [21]. The basic conclusion is that the HR-tree is far more efficient in terms of query processing for *time point* queries while that is not true for *time interval* queries. Also, the HR-tree may result in a rather large structure.

3.3 A set of operations and parameters

[40] discusses a list of specifications to be considered when designing and evaluating efficient STAMs with respect to: (i) data types and datasets supported, (ii) issues on index construction, and (iii) issues on query processing. While the second and third ones mainly address the internal structure of a method and hence should be considered by STAM designers, the first group of specifications highly affect the design of an efficient benchmarking environment since they focus on database characteristics for evaluation purposes. In particular, the specifications that are addressed in [40] with respect to type (i) are the following:

- *Spec 1: on the data type(s) supported.* Appropriate STAMs could support either point or non-point spatial objects. In some cases, point objects could be considered as special cases of non-point objects but this depends on the underlying modeling.
- *Spec 2: on the time dimension(s) supported.* A second classification concerns the time dimension(s) supported, i.e., valid and/or transaction time. Since at least one time dimension should be supported, spatiotemporal databases are classified in *valid-time, transaction-time,* and *bitemporal ones.*
- *Spec 3: on the dataset mobility.* Three cases are addressed, with respect to the motion of objects and the cardinality of the dataset through time, namely *evolving* (i.e., moving objects of a fixed cardinality through time), *growing* (i.e., static objects of varying cardinality through time), and *full-dynamic* (i.e., moving objects of varying cardinality through time) databases.
- *Spec 4: on the timestamp features.* Whether future instances could refer to past timestamps or not leads to a distinction between *chronological* and *dynamic* databases, i.e., collections of objects' instances (o_id, s_i, t_i) that either have or not to obey the rule of consecutive timestamps: $t_{i+1} > t_i$.

In the rest of the paper we study the case of *temporally degenerate* databases that obey the rule of consecutive timestamps, i.e., for each object in the database, the following inequality exists between the timestamp of the current instance t_i and that of the next instance t_{i+1} to be inserted into the database: $t_{i+1} > t_i$. The term *degenerate* refers to the characteristic that the valid time of object instances is identical to their transaction time. That is, an object is valid as long as it exists in the database. The problem that arises when no such rule exists[3] is clarified through the following example: Consider that two instances (o_id, s_i, t_i) and (o_id, s_j, t_j) of an object o have been inserted into the database (without a loss of generality, we assume that $t_i < t_j$) and no instance (o_id, s_k, t_k) exists, such that $t_i < t_k < t_j$. Hence $[t_i, t_j)$ is the valid (and transaction) time of instance i. Let now assume that a new instance (o_id, s_l, t_l) is inserted into the database, such that $t_i < t_l < t_j$. Due to that action, (a) the valid time of instance i has to be changed from $[t_i, t_j)$ to $[t_i, t_l)$ and (b) the validity interval of the new instance l has to be set to $[t_l , t_j)$. No straightforward support for those operations exists in current STAMs and, therefore, we currently leave that case out of study. Note however, that this assumption is not made in the area of bitemporal databases [31]. Indeed in bitemporal access structures the rule is that, by definition, only transaction is monotonically increasing as discussed above. However, adding spatial features to bitemporal data is still an open area for research.

[3]Applicable to valid-time only since transaction-time always obeys that rule.

3.3.1 User requirements

Three *(Spec1 to Spec3)* out of the above four specifications are orthogonal to each other. On the other hand, only the *chronological* case of *Spec4* is supported in this study, as declared earlier, and, as a result of that, we currently treat *transaction-* and *valid-* time under a uniform platform. Hence, we distinguish among 12 different database families (e.g. a *point* plus *transaction-time* plus *evolving* plus *chronological* database) according to the following options:

- *Spec1: point vs. region* database,
- *Spec2: transaction- (or valid-) vs. bitemporal* database,
- *Spec3: evolving vs. growing vs. full-dynamic* database,
- *Spec4: chronological* database.

In order for the user of a benchmarking environment to generate a synthetic dataset, he/she should be able to (a) select one among the above database options and, then (b) tune the cardinality of the dataset and an appropriate set of parameters and distributions.

A fundamental issue on generating synthetic spatiotemporal datasets is the definition of a *complete set of parameters* that control the evolution of spatial objects. Towards this goal, we first address the following three operations:

- *duration of an object instance*, which involves change of timestamps between consecutive instances,
- *shift of an object*, which involves change of spatial location (in terms of center point shift), and
- *resizing of an object*, involves change of an object's size (only applicable to non-point objects).

In a more general case, the latter one could be regarded as *reshaping of an object*, as not only size but also shape could change. However, as the MBR is the most common approximation used by indices, we only consider that case, and thus shape changes are not an issue.

A description of each operation follows. In particular, the goal to be reached is the calculation of the consecutive instances $(o\_id,\ s_i,\ t_i)$ of an object o (recall the definition in Section 3.1) starting from an initial instance $(o\_id,\ s_1,\ t_1)$. We also assume that the spatial workspace of interest is the unit square $[0,1)^2$ and time varies from 0 to 1 (i.e., the unit interval). For illustration reasons, in Figure 3.4 we visualize four instances of a time-evolving two-dimensional region object o and the corresponding projections on spatial plane and temporal axis, respectively.

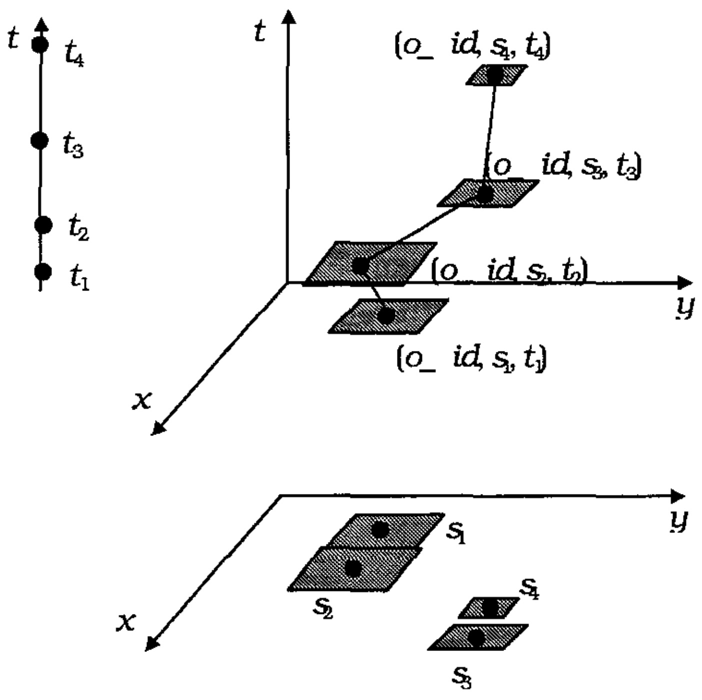


Figure 3.4: Consecutive instances of a time-evolving object o and the corresponding projections

3.3.2 Parameters involved

The *shift*, the *duration*, and the *resizing* of an object's instance are represented by the functions:

duration *(o_id,interval,current_timestamp,new_timestamp)*
shift *(o_id,*Δ*center[],current_spacestamp_center,new_spacestamp_center)*
resizing *(o_id,*Δ*extent[],current_spacestamp_extent[],new_spacestamp_extent[])*

which calculate *new_timestamp* (a numeric value), *new_spacestamp_center* (a 2-dimensional point), and *new_spacestamp_extent[]* (an array of 2 intervals), respectively, of an object *o_id*, as functions of the respective current values and three parameters, namely *interval*, Δ*center[]*, and Δ*extent[]*, respectively.

As an example, consider the object illustrated in Figure 3.4 and its initial position (*o_id*, s_1, t_1). Each consecutive spacestamp s_i and timestamp t_i (i = 2, 3, 4) depends on the previous one, s_{i-1} and t_{i-1} respectively, with respect to the following formulae: $t_i = t_{i-1} +$

interval$_i$, $S_{i.center.x} = S_{i-1.center.x} + \Delta\text{center}_i.\text{x}$, and $S_{i.extent.x} = S_{i-1.extent.x} + \Delta\text{extent}_i.\text{x}$.

In summary, Table 3.1 lists the parameters of interest and their corresponding domains. All parameters should follow a (user-defined) distribution, such as the ones we discuss in the following subsection.

Table 3.1: Parameters for generating time-evolving objects

Parameter	Type	Domain
interval	number	(0..1)
Δcenter[]	2-dimensional vector	$(-1..1)^2$
Δextent[]	2-dimensional vector	$(-1..1)^2$

3.3.3 Distributions

A benchmarking environment should support a wide set of well-established initial data distributions. Figure 3.5 illustrates three two-dimensional initial distributions, namely the uniform, the gaussian, and the skewed one.

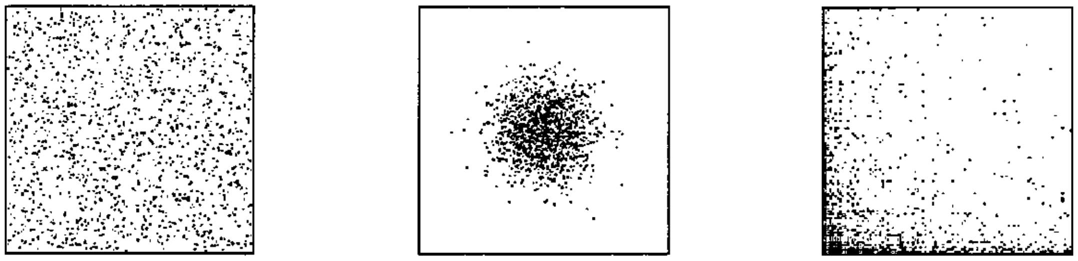


Figure 3.5: Basic statistical distributions in two-dimensional space

In addition to the initial spatial distributions, there are several other parameters that require some kind of statistical distribution, especially those mentioned above (Δ*center[]*, *interval*, and Δ*extent[]*). Through careful use of, possibly different, distributions for the above parameters one may simulate several interesting scenarios, for instance, using a random distribution for the Δ*center[i]* as well as for the *interval*, all objects would move equally fast (or slow) and uniformly on the map; whereas using a skewed distribution for the *interval* one would obtain a relatively large number of slow objects moving randomly, and so on. Also, by properly adjusting the distributions for each Δ*center[i]*, one may control the direction of the objects movement. For instance, by setting Δ*center[i]* = Uniform(0,1) $\forall$ *i*, one would obtain a scenario where the set of objects eventually converge to the upper-right corner of the unit workspace, irrespectively from the initial distributions, but using the "adjustment" approach (see subsection 3.4.1). Similarly, if one wants the objects moving towards some specific direction (e.g. East), he/she can adjust $\Delta center$ and put lower and upper bounds for the center's generated value, as will be discussed in detail in the following section.

Among the distributions supported and illustrated in Figure 3.5, the uniform distribution only requires the minimum / maximum values while the other ones require extra parameters to be tuned by the user. In particular, the gaussian distribution needs *mean* and *variance* parameters as input and the skewed distribution needs a parameter to be declared, which controls the "skewedness" of the distribution.

In the following section, we adopt the issues discussed earlier in order to present an algorithm that generates synthetic spatiotemporal datasets for benchmarking purposes.

3.4 The GSTD algorithm

We propose an algorithm, called *Generate_Spatio_Temporal_Data* (GSTD), for generating time-evolving (i.e., moving) point or rectangular objects. For each object *o_id*, GSTD generates tuples of the format: *(o_id, t, p_l, p_u)*, where t is the timestamp and p_l *(p_u)* is the lower (upper) coordinate point of the spacestamp. The GSTD algorithm is illustrated in Figure 3.6.

3.4.1 Description of the algorithm

GSTD gets several user-defined parameters as input:

- N and D correspond to the initial cardinality and density (i.e., the ratio of the sum of the areas of data rectangles over the workspace area) of the dataset,
- *starting_id* corresponds to the initial id number of the objects,
- *numsnapshots* corresponds to the time resolution of the workspace,
- *min_t* and *max_t* correspond to the domain of the *interval* parameter,
- *min_c[]* and *max_c[]* correspond to the domain of the Δ*center[]* parameter,
- *min_ext[]* and *max_ext[]* correspond to the domain of the Δ*extent[]* parameter, and generates several tuples for each object, according to the following procedure:

"Each object is initially active and, for each one, new instances are generated as long as their timestamp $t < 1$; when all objects become inactive, the algorithm ends".

During the initialization phase (lines 01-04), all objects' instances are initialized, such that their center points are randomly distributed in the workspace, based on the *distr_init()*

distribution, and their extensions are either set to zero (in case of point datasets) or calculated according to *extent(N,D)* routine with respect to the input N and D parameters (in case of non-point datasets).

During the main loop phase (lines 06-27), each new instance of an object is generated as a function of the existing one and the three parameters (*interval*, Δ*center[]*, Δ*extent[]*). Then, invalid instances (i.e., those with coordinates located outside the predefined workspace) can be manipulated in three alternative ways as described below.

In order for a new instance to be generated, the *interval*, Δ*center[]* and Δ*extent[]* values are calculated by calling an *RNG(distr(),min,max)* routine, i.e., a random number generator that generates random numbers between *min* and *max* following a predefined *distr*, which is a statistical distribution, such as the ones discussed in subsection 3.3.3.

The *print_instance* function checks whether the current instance of an object has a timestamp value greater than or equal to the value in *next_snapshot*. If so, the coordinates of the instance (given by the *old_instance* variable) before the current instance are printed, using the apropriated timestamp (which depends on the *next_snapshot* variable). In addition, the value of the *next_snapshot* variable is properly adjusted. Otherwise, the current instance is not output.

Obviously, it is possible that a coordinate may fall outside the workspace; GSTD manipulates *invalid* instances according to one among three alternative approaches:

- the *'radar'* approach, where coordinates remain unchanged, although falling beyond the workspace,
- the *'adjustment'* approach, where coordinates are adjusted (according to linear interpolation) to fit the workspace, and
- the *'toroid'* approach, where the workspace is assumed to be toroidal, as such once an object traverses one edge of the workspace, it enters back in the "opposite" edge.

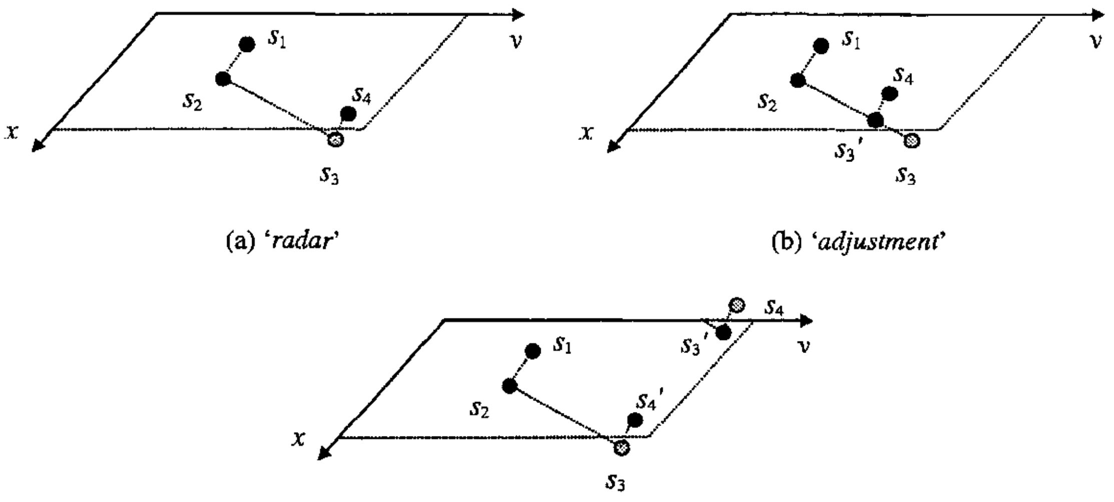
In the first case, the output instance is appropriately flagged to denote that invalidity but the next generated instance is based on that. On the other hand, in the other two cases, it is the modified instance that is stored in the resulting data file and used for the generation of the next one. Notice that in the *'radar'* approach, the number of objects present at each time instance may vary.

The three alternative approaches are illustrated in Figure 3.7 for the example of Figure 3.4. For simplicity, only the centers are illustrated; black (grey) locations represent valid (invalid) instances. In the example of Figure 3.7a, the *'radar'* fails to detect s_3, hence it is not stored but the next location s_4 is based on that. Unlike *'radar'*, the other two approaches calculate a valid instance s_3' to be stored in the data file which, in turn, is used by GSTD for the generation of s_4. It is interesting to watch the behavior of s_4 in Figure 3.7c,

```
Generate_Spatio_Temporal_Data algorithm
Input: values N, starting_id, numsnapshots, D, min_t, max_t
 arrays min_c[], max_c[], min_ext[], max_ext[]
 distributions distr_init(), distr_t(), distr_c(), distr_ext()
Output: instance (id, t, lower_left_point, upper_right_point), validity_flag
begin
01 for each id in range [starting_id .. N+starting_id] do
       /* initialization phase */
02     Set t = 0, center[] = RNG(distr_init(), 0, 1), extent[] = extent(N, D)
03     Set active = TRUE
04 end-for
05 Set step = 1 / numsnapshots
06 for each id in range [starting_id .. N+starting_id] do  /* loop phase */
07     Set next_snapshot = step
08     while active do
           /* calculate delta-values and new instances */
09         Set interval = RNG(distr_t(), min_t, max_t)
10         Set delta_center[] = RNG(distr_c(), min_c[], max_c[])
11         Set delta_extent[] = RNG(distr_ext(), min_ext[], max_ext[])
12         Set old_instance = instance
13         update_instance(instance)
           /* check instances and output */
14         if t > 1 then
15              active = FALSE
16              print_instance(old_instance,current[i],next_snapshot)
17         else //check instance validity and output
18              Set validity_flag = valid(instance)
19              if validity_flag = FALSE and approach <> 'radar' then
20                   adjust_coords(instance, approach)
21              end-if
22              if t > next_snapshot then
23                   print_instance(old_instance,current[i],next_snapshot)
24              end-if
25         end-if
26     end-while
27 end-for
end.
```

Figure 3.6: The GSTD algorithm

where the calculated location finally stored (s_4') is actually identical to that in Figure 3.7a, as the effect of two consecutive calculations for s_3' and s_4'.



(a) '*radar*' (b) '*adjustment*' (c) '*toroid*'

Figure 3.7: Two-dimensional time-evolving spatial objects

3.4.2 Examples of generated dabasets

As mentioned earlier, real world examples of (point or region) spatiotemporal datasets include trajectories of humans, animals, or vehicles, e.g. detected by a global positioning system (GPS), digital simulations of flights or battles, weather forecast and monitoring of fire or hurricane fronts. For instance, detecting vehicle motion by GPS and storing the whole trajectory in a database is a typical every day life example. However, different motion scenarios correspond to different datasets which an efficient structure should be evaluated on. Random versus biased direction, fast versus slow motion are some of the parameters that result to totally different applications.

In this subsection, in order to simulate some of those scenarios, we present six example datasets consisting of point or rectangle objects generated by GSTD. For all files the following parameters were set: $N = 1000$, $D = 0$ or 0.5 (for points or rectangles, respectively), *numsnapshots* = 100. Illustrated snapshots correspond to t = 0, 0.25, 0.50, 0.75, and 1. Table 3.2 presents the non-fixed input parameters and the generated snapshots for each file.

Scenarios 1 and 2 follow the *'toroid'* and *'radar'* approach, respectively, to manipulate invalid instances, while scenarios 3 through 6 follow the *'adjustment'* approach.

Scenarios 1 and 2 illustrate points with initial gaussian distribution moving towards East and NorthEast, respectively. In the former case, where the *toroidal* world model was used, when the points traverse the right edge, they enter back in the left side of the map. Notice that to force the points moving to the East, Δ*center[y]* $= 0$ and Δ*center[x]* > 0. In the latter case, where the 'radar' approach is simulated, the points move towards NorthEast and some of them fall beyond the upper-right corner (some quite early due to their speed), in fact some points move beyond the map. Notice that since Δ*center[]* is always > 0, those points will never reappear in the map.

Scenario 3 illustrates the initially skewed distribution of points and the movement towards NorthEast. As the 'adjustment' approach was used, the points concentrate around the upper-right corner. Scenario 4 includes rectangles initially located around the middle point of the workspace, which are moving and resizing randomly. The randomness of *shift* and *resizing* is guaranteed by the *Uniform(min,max)* distribution used for Δ*center[]* and Δ*extent[]*, where *abs(min)=abs(max)* > 0.

Finally, scenarios 5 and 6 exploit the *speed* parameter of a moving dataset as a function of the GSTD input parameters. By increasing (in absolute values) the *min* and *max* values of Δ*center[]*, a user can achieve 'faster' objects while the same could happen by decreasing the *max_t* value that affects *interval*. Thus, the speed of the dataset is considered to be a *meta-information* since it could be derived by the knowledge of the primitive parameters. Similarly, the direction of the dataset can be controlled, as presented in scenarios 1 through 3.

Alternatively, if the user's application makes necessary the conjunction of two (or more) scenarios, as for instance, a population of MBRs with only a small percentage of them moving towards some direction and the rest ones being static, two individual scenarios can be generated according to the above by properly setting the two *starting_id* input parameters and then merged, which is a straightforward task. Bottomline, by properly adjusting the parameters of Table 3.1, one can yield a scenario that fits his/her needs.

3.5 Discussion and Related Work

An alternative straightforward algorithm for generating *N* time-evolving objects would include the calculation of the spacestamp of each object at each snapshot, thus leading to an output consisting of $T = N$. *numsnapshots* tuples. Our approach outperforms that since it outputs a limited number *T'* of tuples ($T' \ll T$), i.e., the necessary ones in order to reproduce the dataset motion.

However, a fundamental question arises: based on the knowledge of two instances (*o_id,*

s_i, t_i) and (o_id, s_{i+1}, t_{i+1}) that correspond to consecutive timestamps, what is the location of an object at a time t_j, such that $t_i < t_j < t_{i+1}$? As an example, recall the instances of the object o illustrated in Figure 3.4. The status of its spacestamp between e.g. t_i and t_{i+1} is a "fuzzy" issue. Two alternatives may be followed:

- *projection*: the spacestamp is considered to be static and equal to the one at time t_i,
- *linear interpolation*: the spacestamp is considered to be moving with respect to a start- (at time t_i) and an end- (at time t_{i+1}) position.

Both alternatives find applications in real world; cadastral systems, on the one hand, versus navigational systems, on the other hand, are popular examples. Figure 3.8 illustrates the two alternative scenarios for the example of Figure 3.4.

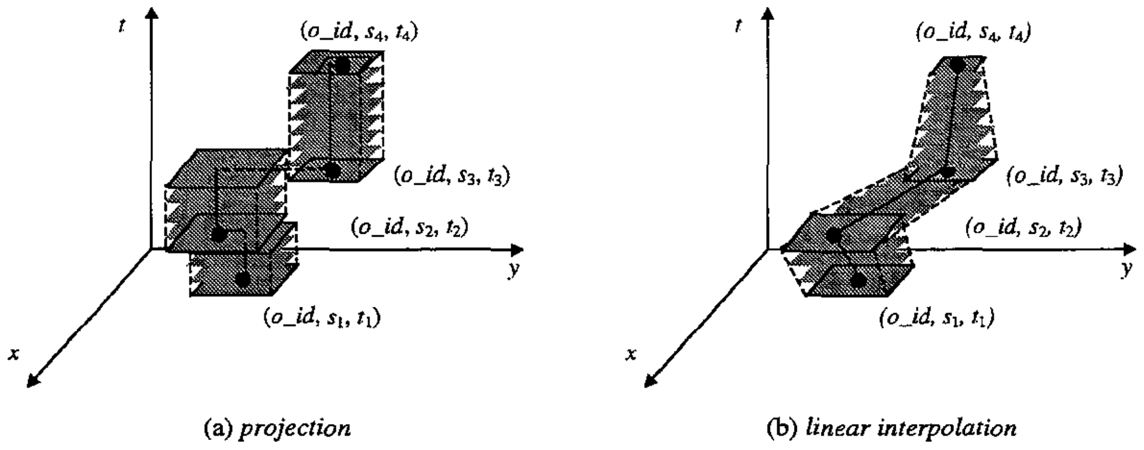


Figure 3.8: Alternative scenarios for the location of an object between two timestamps

In any case, detecting the status of object *o* at a time instance during (t_1, t_2) is an open issue. We argue that the GSTD algorithm proposed earlier is independent of that issue. Actually, it generates a series of instances regardless of such an issue. On the other hand, it is a visualization tool or a STAM construction algorithm that needs to support one or both alternative scenarios. Since in this study we are interested in spatiotemporal databases that follow the rule of consecutive timestamps, the knowledge of both the current and the new instances of an object, as supported by GSTD (line 13), are sufficient to deal with both alternatives.

The need for independent platforms for benchmarking purposes or, in general, experiment management has been already addressed in the past [37, 16]. Such a need arises when a researcher aims to make a 'fair' performance study or experimentation without the dilemma of building his/her own datasets for this purpose. Although extended related work is found in traditional database benchmarks and data generators (e.g. [2, 13]), in the field of spatial databases it is very limited [36, 30, 14]. Moreover, when motion is introduced to support spatiotemporal databases, to our knowledge, no related work exists.

The most relevant to our work is the *'A La Carte'* benchmark [14]. It is a WWW-based tool consisting of a rectangle generator that builds datasets based on user defined parameters (cardinality, coverage, coordinates' distributions) and an experimentation module that runs experiments on either user built or stored sample datasets (including parts of the Sequoia 2000 storage benchmark [37]). The module is actually a *spatial join* performance evaluator that supports several spatial join strategies.

3.6 Conclusion and Future Work

STDBMS require appropriate indexing techniques on spatiotemporal data. Although conceptually the problem seems to be easy to solve, several issues arise when one attempts to adopt a spatial indexing method to organize time-evolving objects by just adding an extra dimension for time. Therefore, a limited number of STAMs have been proposed in the literature as briefly surveyed in Section 3.2.

The effort towards the design and implementation of a benchmarking environment in order to provide performance comparison of STAMs leads to the need of collecting a variety of appropriate synthetic and real spatiotemporal datasets. However, in accordance to the design of efficient methods, generating efficient synthetic datasets is not a straightforward extension of generating spatial data, such as the ones that have been thoroughly used for experimental purposes in the spatial database literature. At a first step, several specifications that identify the type of the dataset have to be addressed and, at a second step, a set of parameters and corresponding distributions have to be tuned by the user. More specifically, we have discussed three operations, namely *duration of an object instance, shift* and *resizing of an object* (the latter one applicable to non-point objects) and derived a set of three parameters, namely *interval*, Δ*center*, and Δ*extent*, which control the evolution of a spatial object through time in satisfactory terms.

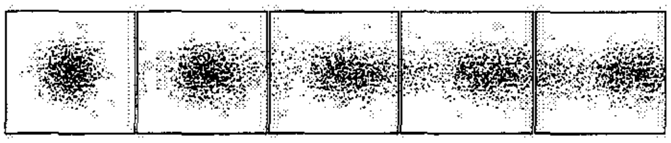
Based on those parameters, we have designed and implemented the GSTD algorithm that generates sets of moving points or rectangles according to users' requirements, thus providing a tool that simulates a variety of possible scenarios. Some of those scenarios have been illustrated and discussed in Section 3.4. GSTD also includes alternative methodologies to support *invalid* instances, i.e., those with coordinates falling outside the workspace.

This study continues the work initiated in [40] towards a full and interactive support tool for designing, implementing, and evaluating access methods for the purposes of STDBMS. We are currently working on a WWW environment to make GSTD available to all researchers through the Internet (mirror sites: *http://www.dblab.ece.ntua.gr/~theodor/GSTD/* and *http://www.dcc.unicamp.br/~mario/GSTD/*). We are also investigating some additional functionality on GSTD. For example, users may want to specify a movement flow to a specific point p in the workspace. Although, given the target p, it is not a complicate task, it is a specific implementation to a specific scenario. We currently study the parameterization of such specific scenarios by permitting GSTD input parameters to be (user-defined) functions rather than fixed values. Such an extension will enhance GSTD flexibility to simulate a variety of real applications.

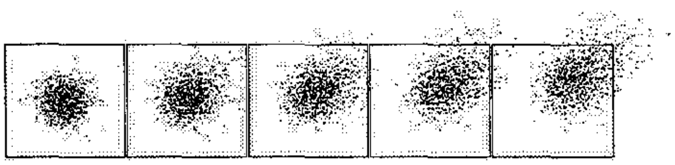
Acknowledgements

Mario A. Nascimento was partially supported by CNPq (Process No. 300208/97-9) and MCT/PRONEX's project "Advanced Information Systems". He is also with CNPTIA/ Embrapa. Jefferson R. O. Silva was supported by FAPESP (Process No. 97/11205-8). Yannis Theodoridis was partially supported by the EC funded TMR project "CHOROCHRONOS: A Research Network for Spatiotemporal Database Systems", contract number ERBFMRX-CT96-0056.

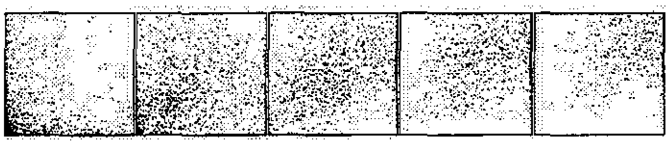
Distributions of Parameters	Snapshots

`distr_init = Gaussian(0.5,0.1)`
`interval = Gaussian(0,0.5)`
`Δcenter[x] = Uniform(0,0.3)`
`Δcenter[y] = Uniform(0,0)`
`Δextent[x] = Uniform(0,0)`
`Δextent[y] = Uniform(0,0)`
	

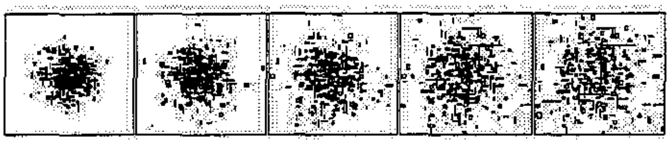
scenario 1: points moving from center to East (*'toroid'* approach)	

`distr_init = Gaussian(0.5,0.1)`
`interval = Gaussian(0,0.5)`
`Δcenter[x] = Uniform(0,0.4)`
`Δcenter[y] = Uniform(0,0.4)`
`Δextent[x] = Uniform(0,0)`
`Δextent[y] = Uniform(0,0)`
	

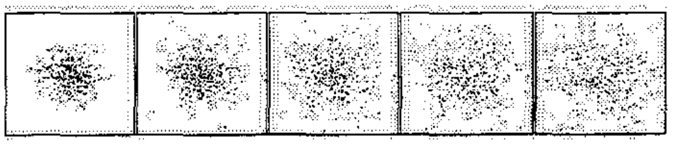
scenario 2: points moving from center to NorthEast (*'radar'* approach)	

`distr_init = Skewed(1)`
`interval = Gaussian(0,0.2)`
`Δcenter[x] = Uniform(0,0.3)`
`Δcenter[y] = Uniform(0,0.3)`
`Δextent[x] = Uniform(0,0)`
`Δextent[y] = Uniform(0,0)`
	

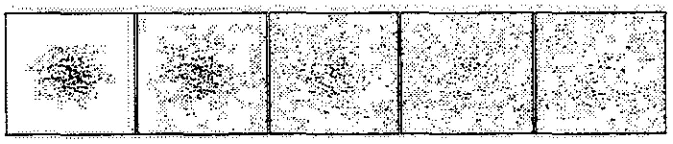
scenario 3: points moving from SouthWest to NorthEast	

`distr_init = Gaussian(0.5,0.1)`
`interval = Gaussian(0,0.5)`
`Δcenter[x] = Uniform(-0.2,0.2)`
`Δcenter[y] = Uniform(-0.2,0.2)`
`Δextent[x] =Uniform(-0.01,0.01)`
`Δextent[y] =Uniform(-0.01,0.01)`
	

scenario 4: rectangles moving (and resizing) randomly	

`distr_init = Gaussian(0.5,0.1)`
`interval = Gaussian(0,0.5)`
`Δcenter[x] = Uniform(-0.2,0.2)`
`Δcenter[y] = Uniform(-0.2,0.2)`
`Δextent[x] = Uniform(0,0)`
`Δextent[y] = Uniform(0,0)`
	

scenario 5: points moving randomly (low speed)	

`distr_init = Gaussian(0.5,0.1)`
`interval = Gaussian(0,0.5)`
`Δcenter[x] = Uniform(-0.4,0.4)`
`Δcenter[y] = Uniform(-0.4,0.4)`
`Δextent[x] = Uniform(0,0)`
`Δextent[y] = Uniform(0,0)`
	

scenario 6: points moving randomly (high speed)	

Table 3.2: Example files generated by GSTD

Capítulo 4

Testes de desempenho da HR-tree

Prólogo

O terceiro artigo apresenta um estudo comparativo entre a HR-tree [27], a 3D R-tree[43] e a 2+3 R-tree[20]. O artigo apresenta inicialmente uma motivação e uma revisão bibliográfica sobre estruturas de acesso à dados espaço-temporais. As três estruturas são descritas de forma resumida, mostrando suas características. Em seguida são descritos os passos realizados para geração dos dados utilizados no teste. O algoritmo descrito no capítulo 3 foi utilizado. O artigo apresenta em seguida, de forma detalhada, os resultados obtidos na comparação.

Como é comum na área de indexação, três fatores foram avaliados nos testes:

- tamanho do arquivo de índice gerado
- número de páginas de disco acessadas na construção do índice
- número de páginas de disco acessadas em consultas

Verificou-se que a HR-tree foi muito superior às outras duas em consultas espaciais em um ponto específico no tempo. Para consultas espaciais em intervalos de tempo, a HR-tree é superior apenas para consultas com intervalos de tempo pequenos. A medida que o tamanho do intervalo aumenta, a HR-tree perde sua vantagem em relação às outras duas estruturas. No fator tamanho de índice, a HR-tree foi a que gerou maiores arquivos de índice, muito acima das outras duas, que geraram arquivos de tamanho próximos uma da outra.

Para tentar diminuir o tamanho do índice gerado, foi explorada no artigo uma outra abordagem, na qual objetos são atualizados não mais a cada instante no tempo, mas a cada t instantes no tempo. Embora nessa abordagem as respostas às consultas tenham que ser filtradas, a HR-tree continua como a melhor estrutura para consultas para pontos no tempo e intervalos pequenos de tempo.

Uma versão anterior desse artigo foi publicado como relatório técnico IC-98-34 [1]. O artigo (na sua versão atual apresentada neste capítulo) foi submetido para publicação no *6th International Symposium on Spatial Databases (SSD).*

[1]Disponível em http://www.dcc.unicamp.br/ic-tr-ftp/1998/98-34.ps.gz

Access Structures for Moving Points

Mario A. Nascimento Jefferson R. O. Silva Yannis Theodoridis

Institute of Computing
State University of Campinas
PO Box 6041
13083-970 Campinas SP BRAZIL
{mario, 972147}@dcc.unicamp.br

Computer Science Division
Dept. of Electrical and Computer Eng.
National Technical University of Athens
Zographou 15773, Athens, HELLAS
theodor@cs.ntua.gr

Abstract

Several applications require management of data which is spatially dynamic, e.g., tracking of battle ships or moving cells in a blood sample. The capability of handling the temporal aspect, i.e., the history of such type of data, is also important. This paper presents and evaluates three temporal extensions of the R-tree, the 3D R-tree, the 2+3 R-tree and the HR-tree, which are capable of indexing spatiotemporal data. Our experiments explore several parameters, and show that the HR-tree is the winner, in terms of query processing cost, for time point and small time interval window queries. On the other hand, its performance deteriorates as the length of the time interval increases. However, the main side effect of the HR-tree is its storage requirement, which is much larger than that of the other approaches. To reduce that, we explore a batch oriented updating approach, at the cost of some overhead during query processing time. To our knowledge, this study constitutes the first extensive experimental comparison of access structures for moving points.

Keywords: spatiotemporal databases, indexing, access structures, benchmarking.

4.1 Introduction

The primary goal of a spatiotemporal database is the accurate modeling of the real world; that is a dynamic world, which involves objects whose position, shape and size change over time [40]. Real life examples that need to handle spatiotemporal data include storage and manipulation of ship and plane trajectories, fire or hurricane front monitor and weather forecast. Geographical information systems are also a source for spatiotemporal data. For instance, in the agricultural domain, land owners as well as government agencies must keep track of crop situation over time to take decision as such to where use pesticides and/or provide financing to farmers. To take another example, consider a series of snapshots taken from a satellite. This type of data, possibly after some treatment, is clearly spatiotemporal data. As yet another example domain, consider the problem of video (or multimedia in general) database management. Objects that appear in each frame can be considered two-dimensional moving objects, upon which one may want to keep track over time or exploit relationships among them. Summarizing, a "database application must capture the time-varying nature of the phenomena they model" [51, Ch. 5]. Spatial phenomena, hence spatial databases, are no exception, therefore spatiotemporal databases should florish as current technology makes it more feasible to obtain and manage such type of data ([9] is a good example of that trend).

Among the many research issues related to spatiotemporal data, e.g., query languages and management of uncertainty [48], we focus on the issue of optimizing access structures, hence speeding up query processing. Despite the fact that there is much work done on the of area of access structures for temporal [31] and spatial data [12], not much has been done regarding spatiotemporal data. This paper deals with this very point.

In particular, we focus on access structures that maitain the whole 'history' of each moving object and are able to answer queries of the type "which objects were located within a specific area at a specific time instance (or during a specific time interval)". This class of access structures currently includes a very limited number of proposals; to our knowledge, there exist only five, namely MR-trees and RT-trees [50], 3D R-trees [43], and more recently, HR-trees [27] and Overlapping Linear Quadtrees [47], based on the popular R-trees [15, 34, 1, 18] and Quadtrees [33]. On the other hand, there exist (also a limited number of) proposals aiming at supporting queries that deal with the future, i.e., "which objects will (certainly or probably) be located within a specific area after (or within) a certain time". This class of access structures usally store current location and some extra information (such as speed and direction) to make safe predictions for the future locations of objects [19, 48]. Although we admit that both applications are of equal importance, in this paper we only consider the former class.

In [40], a set of seven criteria were proposed to characterize spatiotemporal data and access structures in the class of interest:

1. *Data types supported*: whether it supports points and/or regions;

2. *Temporal support*: whether the supported temporal dimension is that of valid time, transaction time or both;

3. *Database mobility*: whether the changes in cardinality or the spatial position of the data items, or both, can change over time;

4. *Data loading*: whether the data set is known a priori or not, whether only updates concerning the current state can be made or whether any state can be updated;
5. *Object representation*: which abstraction (e.g., MBRs) is used to represent the spatial objects.
6. *Temporal treatment*: whether it support special actions such as packing or purging (vacuuming) spatial data as time evolves.
7. *Query support*: whether it is able to process not only pure spatial and temporal queries, but also queries which are spatiotemporal in nature.

After the directions above, and for the purposes of this paper, we assume spatiotemporal data specified as follows:

- The data set consists of two-dimensional points, which are moving within the unit square;
- Updates are allowed only in the current state of the database;
- The timestamp of each point's version grows monotonically following a transaction time pattern, and
- The cardinality of the data set remains fixed as time evolves.

Regarding the indexing structures, no packing or purging of data is assumed. Finally they must provide support to process at least two types of queries: (1) containment queries with respect to a time point; and (2) containment queries with respect to a time interval. By containment query we mean one where, given a reference MBR, all points lying inside such MBR should be retrieved.

Hence, according to the terminology in [40], this paper considers databases of the point/transaction-time/evolving/chronological class. For simplicity, throughout the paper we assume the two-dimensional space, although extending the presented arguments for higher dimension is not problematic. Also, instead of a new access structure, we investigate how to extend a very well known one, namely the R-tree [15, 34, 1, 18].

The remainder of the paper is organized as follows. In Section 4.2 we detail the access structures which we will compare. Next, in Section 4.3, the methodology used to generate spatiotemporal data is discussed. Section 5.4 presents and discusses the experiments we perform regarding space requirements, update and query performance. Finally, the paper is closed with a summary of our findings and directions for future research.

4.2 Spatiotemporal Access Structures

As mentioned earlier, we are aware of only five access structures that consider both spatial and temporal attributes of objects, namely MR-trees and RT-trees [50], 3D R-trees [43], HR-trees [27] and Overlapping Linear Quadtrees [47].

In the RT-tree the temporal information is kept inside the R-tree nodes. This is in addition to the traditional content of the R-tree nodes. On the other hand searching in the RT-tree is only guided by the spatial data, hence temporal information plays a secondary role. As such queries based solely on the temporal domain cannot be processed efficiently, as they would required a complete scan of the database. No actual performance analysis was reported in [50].

The 3D R-trees, as originally proposed in [43], use standard R-trees to index multimedia data. The scenario investigated is that of images and sound in a multimedia authoring environment. In such a scenario it is reasonable to admit that the temporal and spatial bounds of the indexed objects are known beforehand. Aware of that fact the authors proposed two approaches, called the *simple* and the *unified* scheme. In the former one, a two-dimensional R-tree indexes the spatial component of the data set, and an one-dimensional R-tree indexes the temporal component. Query processing is performed using both trees and performing the necessary operations between the two returned answer sets. The latter approach uses a single three-dimensional R-tree and treats time as another spatial dimension. The authors conclude that the advantage of using one or the other approach is a matter of trade-off based on how much often purely spatial or temporal queries are posed relatively to spatiotemporal ones.

The Overlapping Linear Quadtrees, the MR-trees and the HR-trees are all based on the concept of overlapping trees [25]. The basic idea is that, given two trees where the younger one is an evolution[1] of the older one, the second one is represented incrementally. As such only the modified branches are actually stored, the branches that do not change are simply re-used. The Overlapping Linear Quadtrees, as the name implies are based on Quadtrees [33] and as such are not constrained to index only MBRs. The MR-trees and the HR-trees are very similar in nature and we comment on the HR-tree in more details shortly. Indeed, next we discuss the three access structures we investigate in the remainder of this paper.

4.2.1 3D R-tree

The structure we discuss here is based on the 3D R-tree proposed in [43]. The most straightforward way to index spatiotemporal data is to consider time to be another axis, along with the traditional spatial ones. Using this rationale, an object which lies initially at

[1]By evolution we mean a further version based on some changes upon the same data set.

(x_i, y_i) during time $[t_i, t_j)$ and then lies at (x_j, y_j) during $[t_j, t_k)$ can be modeled by two line segments in the three-dimensional space, namely the lines: $\overline{[(x_i, y_i, t_i), (x_i, y_i, t_j))}$ and $\overline{[(x_j, y_j, t_j), (x_j, y_j, t_k))}$, which can be indexed using a three-dimensional R-tree.

This idea works fine if the end time of all such lines are known. For instance consider in the above example that the object moves from its initial position to the new one but is to remain there until some time not known beforehand. All we know is that it lies in its new position until *now*, or *until changed*, no further knowledge can be assumed. The very problem of what *now* or *until changed* means is complex enough by itself (refer to [8] for a thorough discussion on the topic). To make things simpler we assume that *now* (or *until changed*) is a time point sufficiently far in the future, about which there is no further knowledge.

What matters to our discussion is that standard spatial access structures are not well suited to handle such type of "open" lines. In fact, one cannot avoid them. It is reasonable to assume that once the position of an spatial object is known, it is unknown when (and if) it is going to move. As such all current knowledge would yield such open lines, which would render known spatial access structures, e.g., R-trees, of little use. Recently, [5] investigated that problem in the context of temporal databases and proposed appropriate extensions to R*-trees.

One special case where one could overcome such an issue is when all movements are known *a priori*. This would cause only "closed" lines to be input, and thus the above problem would not exist. In the comparisons we make later in the paper using this structure, which we simply refer to as 3D R-tree, we shall make such an assumption. One feature that may favor such an approach is that any R-tree derivative could be used.

4.2.2 2+3 R-tree

One possible way to resolve the above issue is to use two R-trees, one for two-dimensional points, and another one for three-dimensional lines (hence the name 2+3 R-tree). A similar idea has been proposed in [20] in the context of bitemporal databases. In that paper bitemporal ranges with open transaction time ranges were kept under one R-tree (called front R-tree) as a line segment. Whenever a open transaction time range were closed it would become a closed rectangle, which was to be indexed under another R-tree (called back R-tree), after removing the previously associated line segment from the front R-tree. In the 2+3 R-tree whenever the end time of an object's position is unknown it is indexed under a two-dimensional R-tree, keeping the start time of its position along with its id. Note that the original R-tree (or any of its derivatives) keep only the object's id (or a pointer to the actual data record) and its MBR in the leaf nodes. The two-dimensional R-tree used in this approach is thus minimaly modified.

Once the end time of an "open" object's current state (i.e., position) is known, we are able to construct its three-dimensional line as explained above, insert it into the three-dimensional R-tree and delete the existing entry from the two-dimensional R-tree.

Using the example above: from time t_i until the time point immediately before[2] t_j the object is indexed under the two-dimensional R-tree. At time t_j, it moves, as such, (1) the point (x_0, y_0) is deleted from the two-dimensional R-tree, (2) the line $\overline{[(x_0, y_0, t_i), (x_0, y_0, t_j))}$ is input into the three-dimensional R-tree, and, finally, (3) the point (x_1, y_1) is input into the two-dimensional R-tree. Keep in mind that the start time of a point position is also part of the information held along with the remainder of its data.

It is important to note that now, both trees may need to be searched, depending on the time point with respect to which the queries are posed. Similar to the case of the 3D R-tree any of the proposed R-tree derivatives could be used, provided that the leaf nodes of the two-dimensional one is minimally modified as discussed above.

A final remark should be done. The 2+3 R-tree is the dynamic version of the 3D R-tree. That is to say that the two-dimensional R-tree serves the single purpose of holding the current (i.e., open) intervals. Should one know all movements *a priori* the two-dimensional R-tree would not be used at all, hence the 2+3 R-tree would be reduced to the 3D R-tree presented earlier.

4.2.3 HR-tree

The two approaches above have drawbacks. The first suffers from the fact that it cannot handle open-ended lines. The second, while able to overcome that problem, must search two distinct R-trees for a variety of queries. In this section we present the HR-trees [27], which is designed to index spatiotemporal data as classified earlier.

Consider again the example in Section 4.2.1. At time t_i one could obtain the current state (snapshot) of the indexed points, build and keep the corresponding two-dimensional R-tree, repeating this procedure for t_j and t_k. Obviously, it is not practical to keep the R-trees corresponding to all actual previous states of the underlying R-tree. On the other hand it is reasonable to expect that some (perhaps the vast majority) of the indexed points do not move at every timestamp. Consequently R-trees may have some (or many) nodes identical to the previous version. The HR-tree explores this, by keeping all previous states (snapshots) of the two-dimensional R-tree only *logically*.

As an illustration consider the two consecutive (with respect to their timestamps) R-trees in Figures 4.1(a) and (b), which can be represented in a more compact manner as shown in Figure 4.1(c). Though it is just a simple example, it is easy to see that much space could be saved if one could re-use the nodes that did not change from a given state to the next one. Note that with the addition of an array **A** one can easily access the R-tree he/she

[2]We assume, without loss of generality, that the time domain is isomorphic to the rationals.

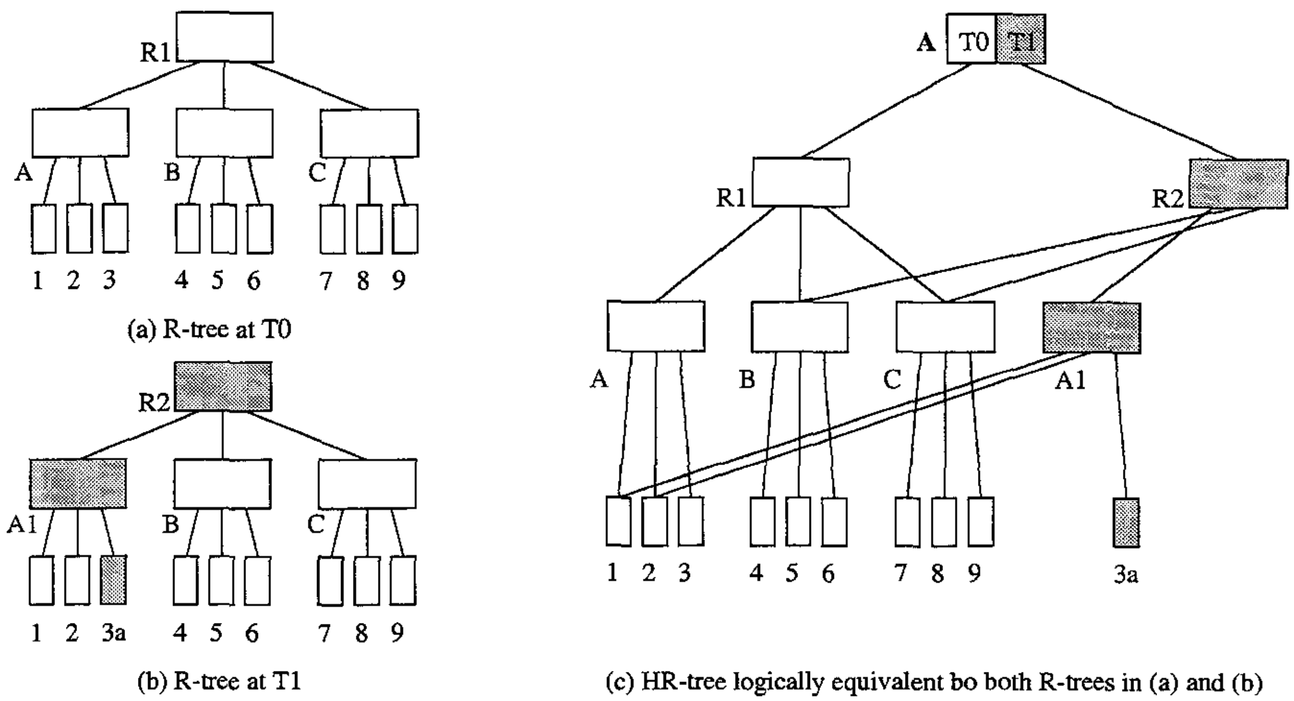


Figure 4.1: Example of the HR-tree approach.

desires. Perhaps most important, is that once the root node of the desired R-tree for a given timestamp is obtained, query processing cost is the *same* as if all R-trees where kept physically.

Notice however that it is desirable to keep the number of newly created branches as low as possible. For that reason some R-tree variants are not suitable to serve as HR-tree's framework, notably, the R$^+$-tree [34] and the R*-tree [1]. In the former the MBRs are "clipped" and one single MBR may appear in several internal nodes, therefore increasing the number of branches to be created in each incremental R-tree. Likewise, the R*-tree, avoids node splitting by forcing entries re-insertion, which is likely to affect several branches, hence enlarging the HR-tree.

Among the other alternatives, we have found the Hilbert R-tree [18] to be very suitable for our purposes and use it as HR-tree's baseline. From now on, unless explicitly mentioned otherwise, we use the term R-tree(s) to refer to the Hilbert R-tree(s) as originally defined.

The overall idea is that upon insertion of a new point version (i.e., its new location) a new branch has to be created with all of its nodes are timestamped with the current timestamp, and then the new point is inserted in this branch's leaf node. All nodes which were not modified at all from last timestamp are simply re-used. For instance, Figure 4.1(c) represent the resulting HR-tree when a new data point is inserted and which were to fall within the leaf node 3 of the R-tree in Figure 4.1(a). A similar idea is used to delete a point from the HR-tree. As commented above, querying the HR-tree is a simple matter of retrieving the appropriate logicall R-tree root through the array **A**. The interested reader can find all HR-tree's algorithms detailed in [28].

4.3 Data Generation

From the seven issues presented in the introduction of the paper, it is quite clear that the first four are mostly related to the data sets while the last three are more related to spatiotemporal access structures.

Hence, with respect to the first four itens, we characterize our data as follows:

- The data type being indexed is that of *points.*
- The supported temporal dimension is that of *transaction time*, the new position of any given point always has a begin-time greater than the end-time of the previous position;
- The data points are *evolving*, i.e., the number of moving points are kept fixed and
- Only the *current state* (snapshot) can be updated.

The access structures in turn are categorized as:

- Able to index points or regions, the last ones would be abstracted by MBRs;
- No purging or vacuuming of data is assumed and
- Queries based on spatial, temporal or spatiotemporal predicates can be processed, although not all with the same efficieny in all structures.

For the data generation itself, we have build GSTD[3] [41], a spatiotemporal data generator where the user can tune several parameters to obtain data sets which fulfills his/her needs. Some of them are: the initial data distribution, the amount of time a point is going to rest at the same location (*interval*), the distance it will move (*shift*) and how it is going to move in the space. For the initial data distribution we have experimented the cases of uniform, gaussian and skewed distributions.

All data is initially generated assuming a two-dimensional unit square. Any point moving out of such square has its coordinates adjusted to fit in the workspace. We also assume that the interval follows a gaussian distribution and the shift and direction follow the uniform one.

[3]URLs: `http://www.dblab.ece.ntua.gr/~theodor/GSTD` and `http://www.dcc.unicamp.br/~mario/GSTD`.

To illustrate how data is generated and how it evolves, Figure 4.2 (4.3) shows an initial data set using the gaussian (skewed) distribution, and four "snapshots" taken after 25, 50, 75 and 100 timestamps. It is easy to note that after 100 timestamps the initial distribution becomes quite close to a uniform one. This feature will allow us to investigate how dependent of the initial spatial data distribution the access structures are. Naturally, data points which were initially randomly distributed, remain randomly distributed as time evolves.

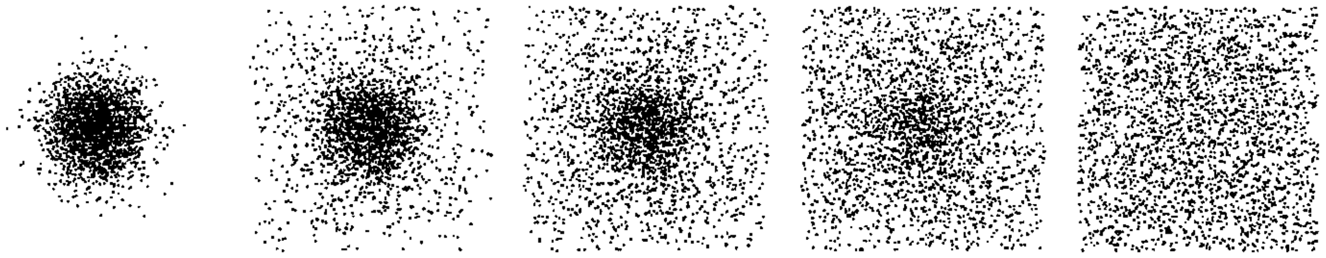


Figure 4.2: Evolution of gaussianly distributed data points.

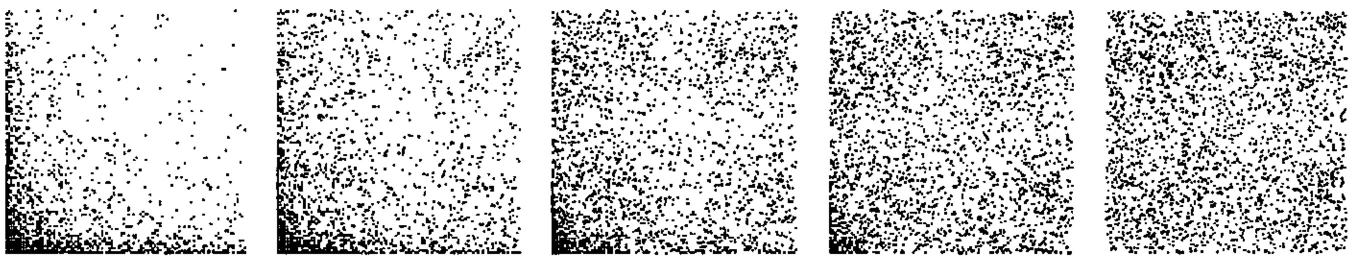


Figure 4.3: Evolution of skewedly distributed data points.

4.4 Performance Comparison

As pointed out above our main concern is to investigate the performance yielded by the 3D R-tree, the 2+3 R-tree and the HR-tree when indexing moving points. Recall that in order to use the 3D R-tree one must know the whole history in advance. While some applications, such as digital battle field, may use *previously recorded snapshots*, their *online versions* always involve the *now* parameter. To overcome that problem, one may use the 2+3 R-tree approach discussed in Section 4.2.2. In any case, we decided to include the 3D R-tree as a yardstick. Both the 3D and 2+3 R-trees are Hilbert R-trees in nature (recall that the HR-tree also uses the Hilbert R-tree as its basis).

Our experiments were performed on a Pentium II 300 Mhz PC running LINUX with 64 Mbytes of core memory. The disk pages, i.e., tree nodes, use 1,024 bytes and all programs were coded using GNU's C++ compiler. Although this paper presents quantitative results

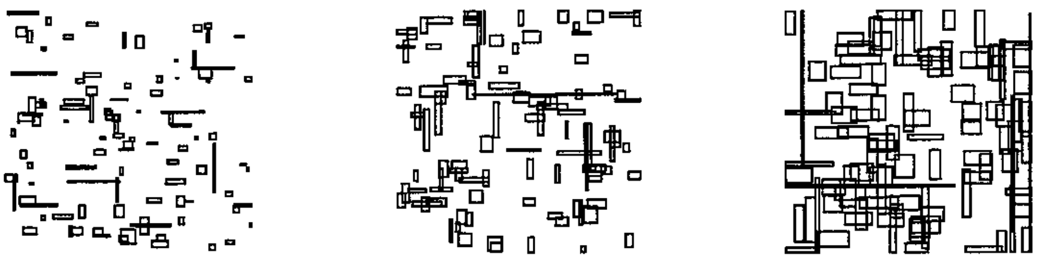

(a) Small MBRs (a) Medium MBRs (a) Large MBRs

Figure 4.4: Query MBRs with centers uniformly distributed.

using only a data set cardinality of 100,000 data points, results using other values can be obtained elsewhere [28]. All objects timestamps fell within the unit time interval [0, 1] with a granularity of 0.01, i.e., 100 distinct and equally spaced timestamps could be identified. Three initial spatial distributions were investigated: uniform, gaussian and skewed. The GSTD parameters were set in such a way that the points could move freely and therefore the final distributions tend to the uniform one (see Figures 4.2 and 4.3). The queries were uniformly distributed, and three different area sizes were used, denoted respectively as small, medium and large MBRs, and shown in Figure 4.4. The difference in the generation of the queries is that they are not points, but MBRs. Recall that we are interested in queries of the type "which are the points contained in a given region at (or during) a time point (interval)". The same set of two-dimensional queries are three-dimensionalized by "adding" a third temporal axis to make containment queries with respect to a time interval. For each time point or interval we ran 100 queries and computed the averages obtained.

Next we show the obtained results regarding storage requirements, index building cost and query processing cost (the last two measured in terms of disk I/O).

4.4.1 Storage Requirements

Figure 4.5 shows how large each of the structures are after indexing the initial 100,000 data points, and all their evolutions, i.e., after 100 timestamps, as a function initial spatial distribution. It is clear that the qualitative behavior of either structure does not seem to depend upon the initial distribution of the spatial data.

One result not shown in the figure is that, as expected, the HR-tree approach does save space when compared to the extreme case of physically storing all (100) R-trees. In fact, the average savings amounted to over 33%.

More interesting however, is the fact that the 3D R-tree and the 2+3 R-tree use roughly the same storage and are much smaller than the HR-tree. This is due to the fact that several branches of the "virtual" R-trees are duplicated in the HR-tree. Indeed, the more dynamic

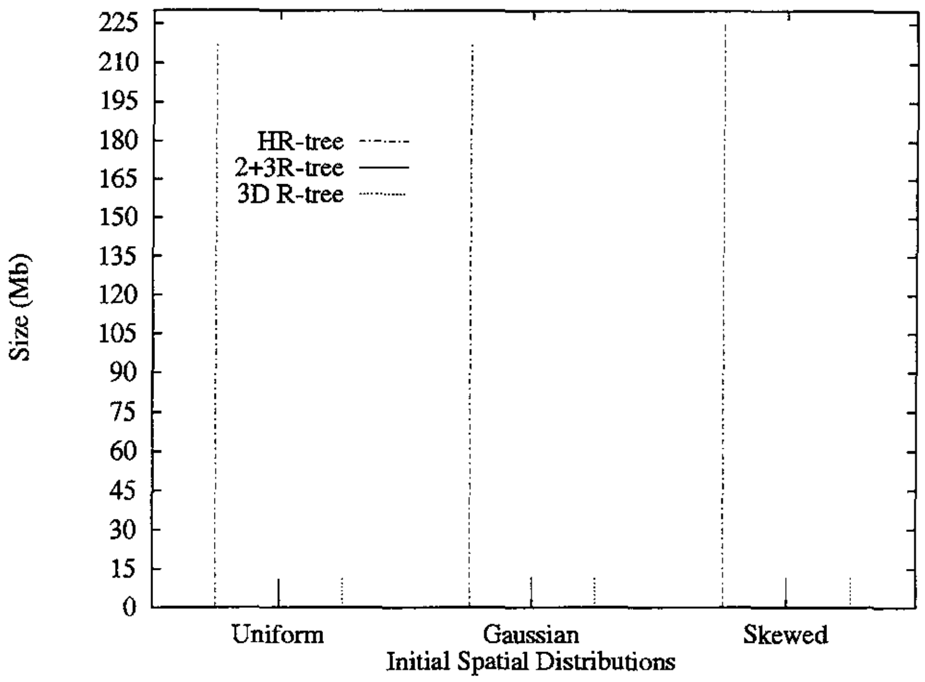


Figure 4.5: Indices' sizes.

the data points, the smaller the savings in the HR-tree with respect to storing all R-trees separately.

In order to alliviate this shortcoming we experimented the following approach. Instead of indexing all movements at the very timestamp when they happen, one could use a batch-oriented approach. The idea is to collect all movements into a buffer and from time to time flush out all of them into the same virtual R-tree. For instance, consider points locations which would originally be indexed under timestamps t, $t+1$ and $t+2$. Using such an approach all such points could be indexed at time $t+2$. The tradeoff in doing this is that one may not be able to query only the time point of interest. In the example just presented querying time point $t+1$ would require retrieving the (virtual) R-tree at time $t+2$, and consequently filter the answer properly. In other words, such batching may yield false-hits. We investigate this trade-off in more details shortly.

Figure 4.6 shows the obtained results when all movements were indexed at their exact timestamp (as in Figure 4.5), and at every two and three timestamps (which we refer to as double and triple approach). The gains were about 30% and 45% respectively. Note that even though the HR-tree became smaller it still remains larger than the other two structures. Fortunately, as we shall see shortly, the imposed overhead due to the false-hits is not likely to be as high (depending on the query size).

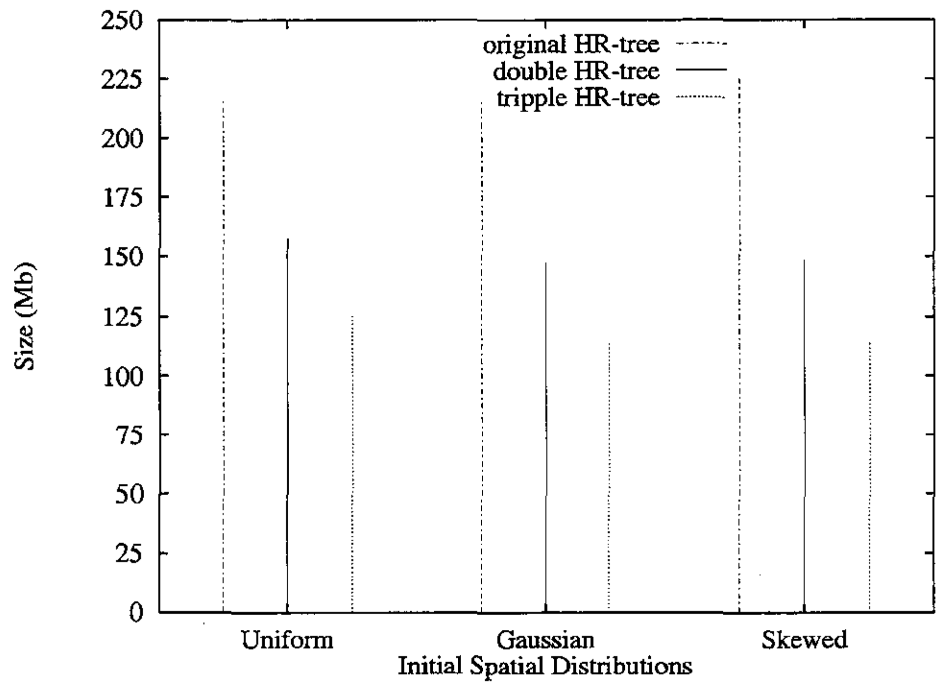


Figure 4.6: HR-tree size using a batch oriented approach.

4.4.2 Index Building Cost

One interesting aspect to consider is the cost (in terms of disk I/Os) needed to construct the indices. Figure 5.3 shows such an information for all three initial distributions after indexing 100,000 data points and their evolutions. As expected the 2+3 R-tree is the one which takes more time (i.e., I/Os) to be built. This is due to the fact that whenever a point moves, there is an insertion (of the new version) and a deletion (of the previous position) on the two-dimensional R-tree, and one insertion of a line (the previous position history) on the three dimensional R-tree. The HR-tree appears as the runner-up as at least one complete branch is updated from one timestamp to another one (assuming at least one point moved). The 3D R-tree on the other hand is the most economical alternative, given that if all point movements are known a priori[4], only one three-dimensional line is inserted per point movement and no deletions occur.

Unlike in the case for the indices sizes, the batching approach yields only some marginal improvements (when used for the HR-tree) regarding update cost.

[4]Recall however the restrictions of such an assumption for several applications, as discussed earlier.

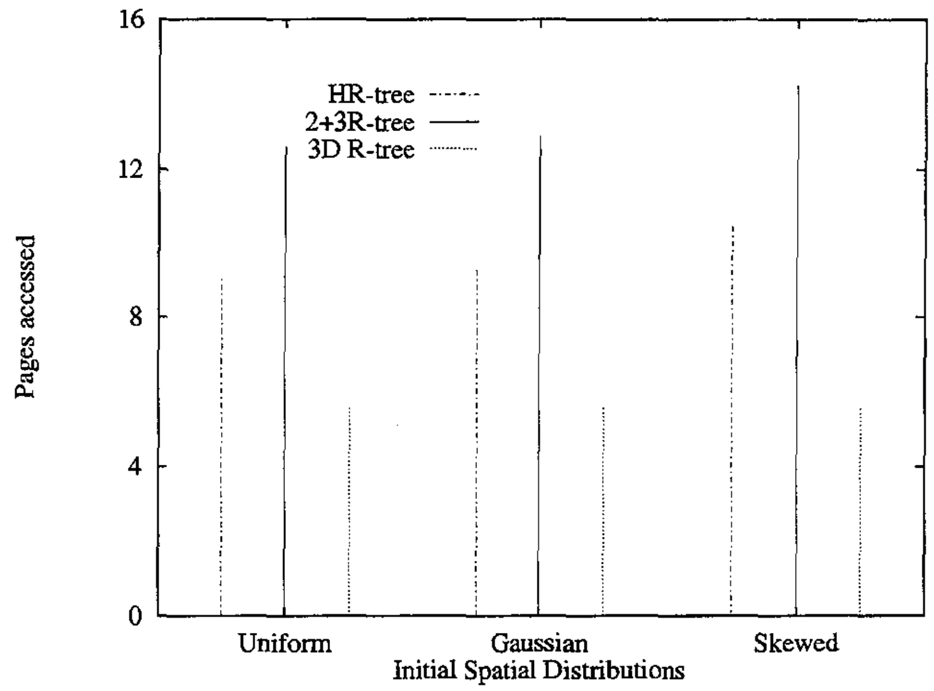


Figure 4.7: Indexing building cost.

4.4.3 Query Processing Cost

There are two cases to consider, one when the query is posed with respect to a specific time point and another when a time interval is considered. We consider each one in turn, starting with the case of a fixed reference time point query. The U, G and S labels in x-axis of the following figures denote the initial Uniform, Gaussian and Skewed spatial distribution respectively. The results shown are the average number of I/Os obtained when querying time points 0, 0.25, 0.5, 0.75 and 1.

Figures 4.8, 4.9 and 4.10 show the results obtained when querying each structure, at the timestamps described above, and using the small, medium and large query MBRs respectively.

Our conjecture that the HR-tree would require substantially smaller query processing time was shown to be true in all cases. When the temporal part of the query is a point, the tree that corresponds to that timestamp (e.g., the R-tree pointed by **A**[T1] in Figure 4.1(c)) is obtained and the query processing is exactly the same one of a range query in a single standard R-tree. On the other hand, for both the 3D and 2+3 R-tree the whole structure is involved in the query processing, thus increasing significantly the query processing cost.

One noteworthy result is obtained when querying the 2+3 R-tree at time 1, i.e., that end of the indexed temporal window. When a query is posed against that time point, the 3D R-tree answer (recall that the 2+3 R-tree is composed by a two and a three-dimensional

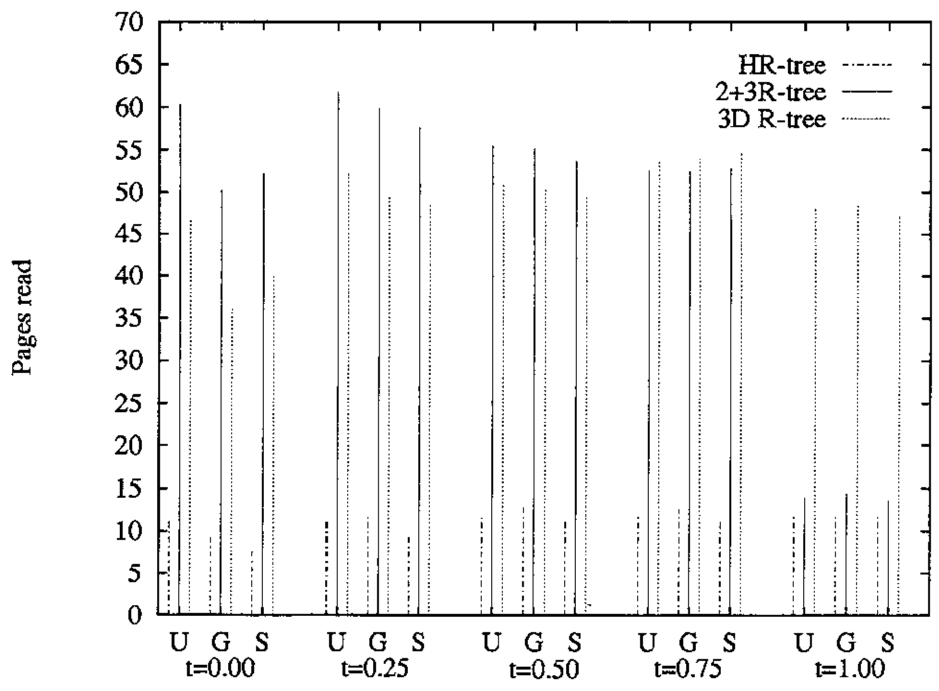


Figure 4.8: Query processing cost, small MBRs.

two R-trees) is empty. This happens because there is no object with an end time equal to 1 and thus all the answers come from the 2D R-tree, where there is always a current version for all indexed points. In fact, when querying time point 0.99, i.e., the indexed timestamp immediately before the last one, the 2+3 R-tree would require over 130% more I/Os than necessary when querying time point 1. Therefore the good performance obtained when querying that particular timestamp is by no means typical. Finally, it also became clear that the size of the query MBR affects all structures equally.

It is now necessary to measure the trade-off posed when one used the double or triple HR-tree approach as discussed earlier (we noted earlier that such approach was able to reduce the HR-tree's size). Figure 4.11 shows the results when querying large query MBRs at time points 0, 0.25, 0.5, 0.75 and 1 (the results were qualitatively the same for small and medium sized queries, and are not shown to save space). The difference in the curves for the original and the double and triple approach represents the number of pages which were read due to the batching, i.e., overhead pages. The average overhead was 17% for small queries, 20% for medium queries and 30% for large queries. It is important to note that despite such overhead the HR-tree still remains the overall faster index structure (compare Figure 4.10 to 4.11). The only exception we noted is the performance of the 2+3 R-tree when querying time point 1, which is, as argued earlier, an atypical result.

Next we proceed to investigate how the structures perform when querying a time interval

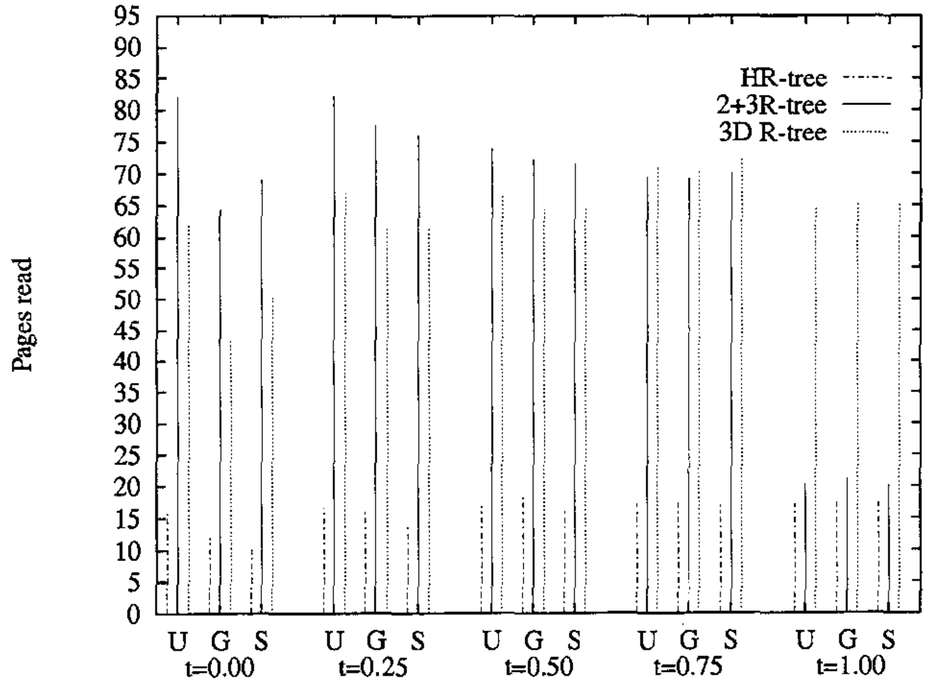


Figure 4.9: Query processing cost, medium MBRs.

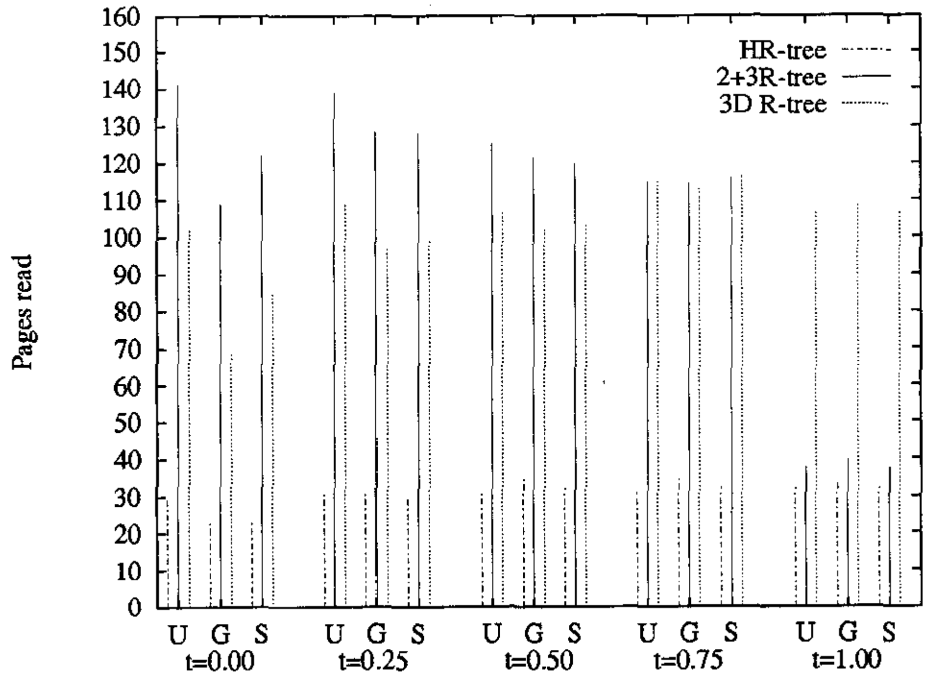


Figure 4.10: Query processing cost, large MBRs.

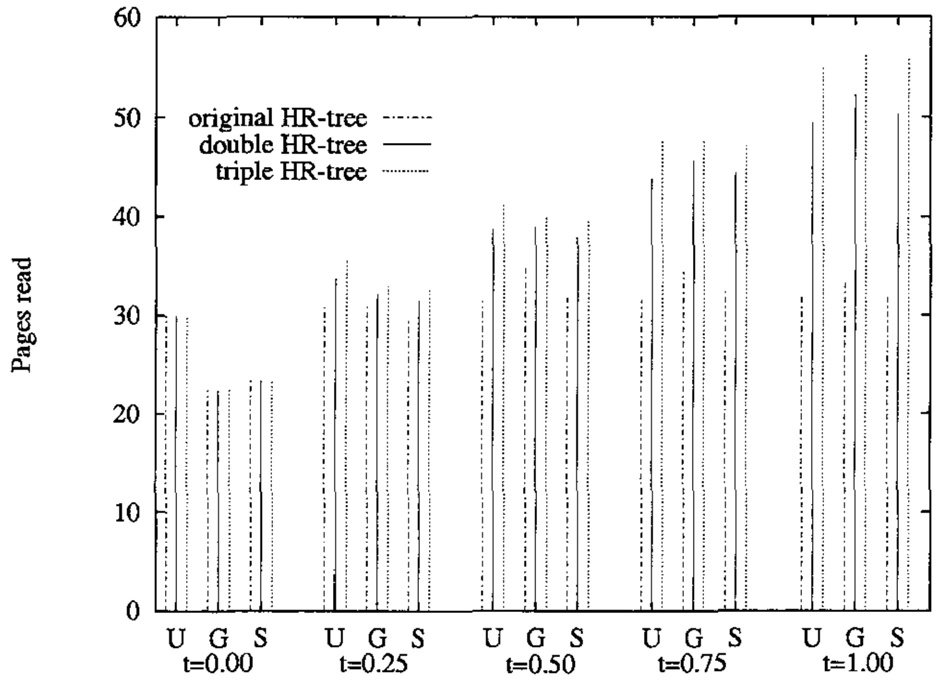


Figure 4.11: Query processing overhead when using the batch oriented approach and large MBRs.

instead of a point. Figures 4.12, 4.13 and 4.14 illustrate the query processing cost when varying the average length of the queried time interval. Such a length is measured in number of consecutive timestamps. That is, we measured the structure's performance when the time interval was up to 10% of the total (unit) time window length. Larger intervals were not investigated as the curves shown already presented a clear trend.

While the HR-tree is well suited to search a time point, i.e. it searches a single (virtual) R-tree, for time intervals it has to traverse as many logical R-trees as many indexed time points are covered by that interval. As a result the HR-tree loses its relative advantage relatively fast with the increase in the queried interval length. In fact, this is worsened by the increase in the query MBR area.

Figure 4.15 shows that the batching approach, as expected, yields some overhead at query processing time which is now harmful (comparing to the performance delivered by the other structures). That is, batching updates does not seem worthwhile for the case of querying time intervals.

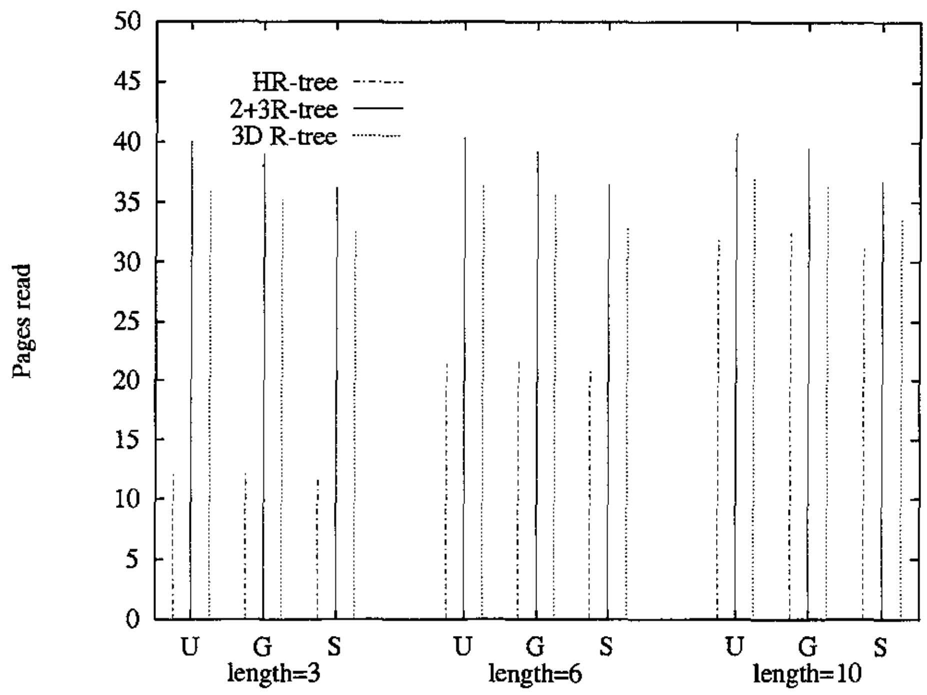


Figure 4.12: Interval query processing cost, small MBRs.

4.5 Summary and Future Research

In this paper we raised the issue that despite the fact that applications dealing with spatiotemporal data are gaining strength, not much has been done regarding implementation and/or extensions of appropriate database management systems. Towards that goal our contribution was to propose and investigate access structures for spatiotemporal data.

To the best of our knowledge it is the first performance study for spatiotemporal access structures, unlike the areas of temporal [31] and spatial [12] indexing methods where an extensive experimentation and comparison is found in the literature. In the particular field that we discuss in this paper, it was only [43] that compared alternative schemes based on R-trees with respect to analytical cost models proposed in [39].

We investigated three R-tree based structures: the 3D R-tree, the 2+3 R-tree and the HR-tree. We have discovered, after several experiments, that:

- The less dynamic the data set, the higher the storage savings yielded by the HR-tree when compared to the ideal (from the query processing perspective) but impractical (in terms of disk storage demand) solution of having all logical R-trees physically stored;
- The 3D and 2+3 R-trees tend to be much smaller than the HR-tree;
- The use of a batching approach at update time is capable of reducing substantially the

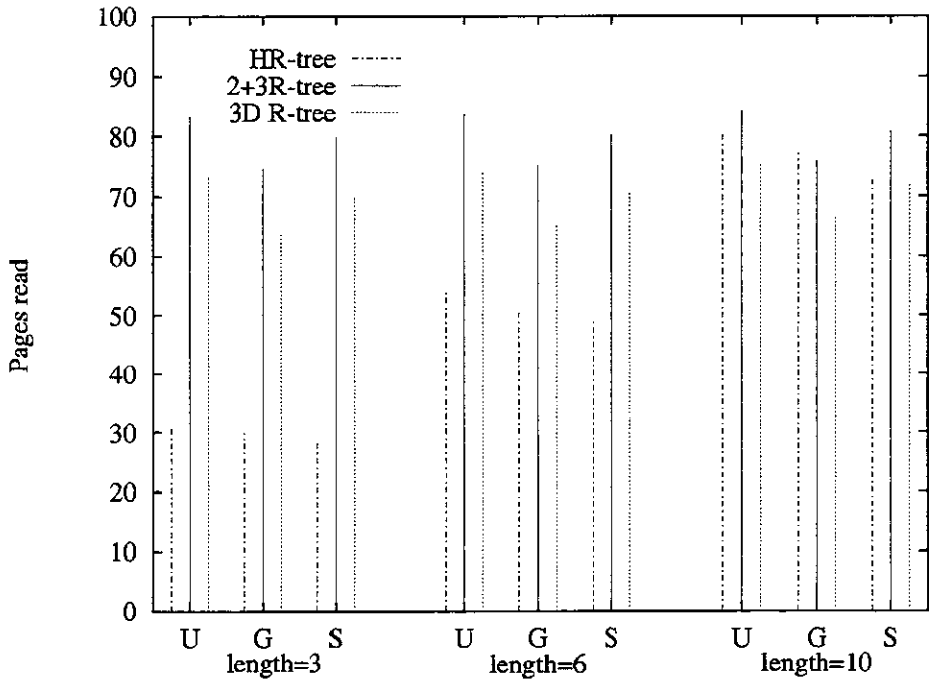


Figure 4.13: Interval query processing cost, medium MBRs.

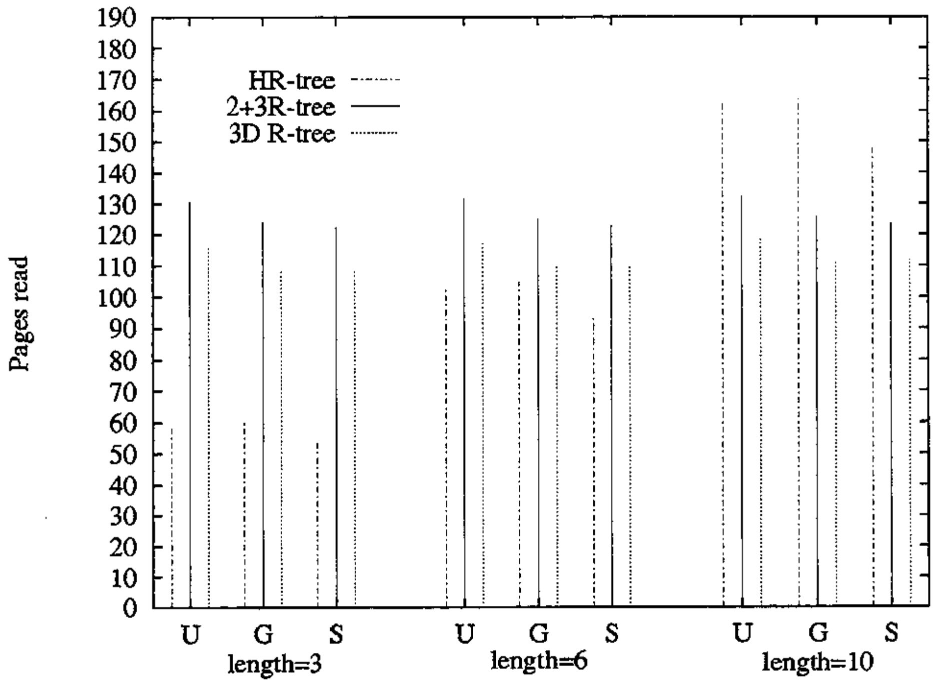


Figure 4.14: Interval query processing cost, large MBRs.

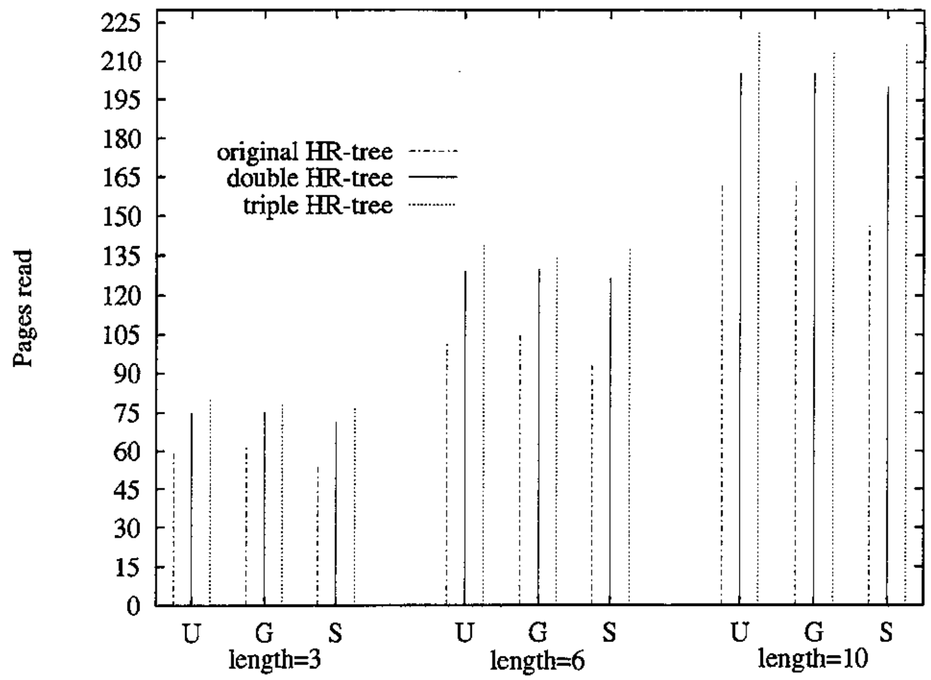


Figure 4.15: Query processing cost, large MBRs.

HR-tree's size, yielding some overhead at query processing time.

- When querying a specific time point the HR-tree offers a much better query processing time than the 3D and 2+3 R-trees. In fact, it offers the same performance as if all logical R-trees were physically stored. The overhead imposed by the batching approach is acceptable as the HR-tree remains the best performer;

- If instead of a time point a time interval is queried, the HR-tree loses its advantage rather quickly with the increase in the length of the queried time interval. In such a case the batching approach overhead for the HR-tree is hardly worthwhile.

Considering that with current technology storage is much less of a problem than time to query data, we consider the HR-tree a good candidate access structure for spatiotemporal data when most queries are posed with respect to a time point or a very short time range. On the other hand, when both short and long time interval queries are probable, the other two structures are also good candidates. We also have to notice that, unlike HR-tree and 2+3 R-tree, 3D R-tree are not capable of supporting on-line spatiotemporal applications that involve the now (or until changed) parameter [8].

The present paper has not dealt specifically with how the above structures behave with respect to movement direction and speed and the use of caching structures and policies. For instance, suppose that instead of spreading in the space, all points move coordinately towards the same direction, how would each access structure support this ? Also a few points may move much faster than the others (or vice-versa), is that a feature that will affect the structures' performance ? These and several other questions are currently being investigated. An issue which needs further research has to do with the abstraction of the data set. In this paper we used data points, but it is easy to foresee that several application domains (e.g., land information systems [49]) may require the management of spatial regions. For the particular case of MBRs the structures proposed in this paper could be used. We conjecture that their strengths and limitations would remain, but further research is needed to confirm such belief.

As a further step, each structure's performance is planned to be analytically explored, in correspondence with the R-tree analysis for selection and join queries that appears in [42]. Both directions, extensive experimentation and analytical work, converge on building a spatiotemporal benchmarking environment consisting of real and synthetic data sets and access structures for evaluation purposes.

Acknowledgements

Mario A. Nascimento was partially supported by CNPq (process number 300208/97-9 and PRONEX's project "SAI: Advanced Information Systems") and is also with CNPTIA - Embrapa. Jefferson R. O. Silva was supported by FAPESP (process number 97/11205-8). Yannis Theodoridis was supported by the EC funded TMR project "CHOROCHRONOS: A Research Network for Spatiotemporal Database Systems", contract number ERBFMRX-CT96-0056. We also thank Timos Sellis for fruitful discussions and comments on earlier drafts of the paper.

Capítulo 5

Aplicação da HR-tree como estrutura de indexação de dados bi-temporais

Prólogo

O quarto e último artigo trata da aplicação da HR-tree em indexação de dados bitemporais. O artigo inicialmente faz um levantamento do problema de indexação em banco de dados bitemporais e dos trabalhos relacionados. Em seguida, falamos da nossa motivação para a aplicação da HR-tree nesse problema e descrevemos como a utilizamos para a indexação bitemporal, assim como a estrutura com a qual foi comparada. A geração dos dados utilizados nos testes é descrita e finalmente o artigo apresenta os resultados obtidos.

Os fatores avaliados nos testes foram os mesmos apresentados no capítulo 4. Basicamente, os resultados obtidos foram: (1) para consultas o desempenho da HR-tree foi bem superior, chegando a ser 80 % mais rápida; (2) também na construção do índice, a HR-tree mostrou-se superior à 2R-tree; (3) já para tamanho do índice gerado, a HR-tree gera índices maiores que a 2R-tree. Utilizando uma política de inserção em *batch*, o tamanho do índice gerado pela HR-tree é reduzido, e o desempenho em consultas é praticamente mantido.

Esse artigo foi publicado como relatório técnico TimeCenter TR-38[1]. Uma versao revisada foi submetida ao *11th IEEE Symposium on Scientific and Statistical Database Management.*

[1]Disponível em http://www.cs.auc.dk/research/DBS/tdb/TimeCenter/TimeCenterPublications/TR-38.ps.gz

An Incremental Index for Bitemporal Databases

Jefferson R. O. Silva Mario A. Nascimento
Institute of Computing
State University of Campinas
P.O. Box 6176
13083-970 Campinas SP BRAZIL
{972147, mario}@dcc.unicamp.br

Abstract

Bitemporal databases record not only the history of tuples in temporal tables, but also record the history of the databases themselves. Indexing structures, which are a critical issue in traditional databases, became even more critical for bitemporal databases. We address this problem by investigating an incremental indexing structure based on R-trees, called the HR-tree, which was originally aimed at spatiotemporal databases. We have found that the HR-tree is much more efficient (up to 80% faster) than previously proposed approaches based on two R-trees when processing transaction time point based queries. As for size, the HR-tree was found to be better suited for medium to large sized batch updates, otherwise it is prone to be quite large.

Keywords: temporal databases, indexing, access structures, R-trees.

5.1 Introduction

Temporal databases have been the object of quite some research regarding many aspects and much has been published in the field [46]. It has been recognized that two dimensions of time need to be supported by a database management system (DBMS) in order to enhance the temporal modeling capabilities of a database. These two time dimensions are the *valid time* (the time range when the fact is true in the modeled world) and *transaction time* (the time range when the fact is logically stored in the database). A third dimension, user-defined time, is also needed for modeling purposes, but need not be supported by the DBMS [17]. A *Bitemporal Database* (2TDB) is thus one which supports both valid time and transaction time. In such a case one is allowed to ask queries based on different (past and possibly future) states of the database and/or tuples.

Our goal in this paper is to investigate the efficiency of an index structure originally designed for spatiotemporal databases, the HR-tree, [27, 28], for indexing 2TBDs. In spatiotemporal databases, one must index not only the spatial extents of objects, which we refer to as *spacestamp*[1], but also their evolutions as time progresses. The current spacestamp is the one stored until it is changed. Hence we assume it is stored until the current point in time, denote by *now*. As such the transaction time of the spacestamps are *now-relative* [5]. On the other hand we assume that the valid-time of those spacestamps are *not now-relative*, i.e., they are known in advance.

As an example of a spatiotemporal scenario that fits the above requirements consider satellite imagery. Each image about a certain area has a well-defined valid time, and each such image is stored until the next one is obtained. Note that even in the case where a new image is not obtained in due time the valid time of the current one is likely to expire due to the very nature of moving objects (e.g., ships, planes, hurricanes) down in Earth. That is, every image has a pre-defined valid time, and a transaction time which is now-relative.

Note that if we simply replace the notion of spacestamps for the notion of regular tuple attributes we obtain a 2TBD. In other words, instead of assigning valid time to spacestamps we do so for a set of regular tuple's attributes. This similarity is the motivation we use to investigate the HR-tree's performance when indexing 2TDBs.

The rest of this paper is organized in the following manner. Section 5.2 reviews previously proposed approaches. Section 5.3 presents briefly the HR-tree structure. In Section 5.4 we discuss how we generated data sets for evaluating the HR-tree and also present the results we obtained. Section 5.5 concludes the paper with a summary and offers directions for further work.

5.2 Related Work

While a reasonable number of papers have been published on the issue of indexing either valid time or transaction time databases, only a handfull have addressed the problem of indexing 2TDBs [31]. In what follows we review some of the approaches proposed recently, most based on R-trees [15]: the M-IVTT [26], the Bitemporal Interval Tree, Bitemporal R-Tree, the 2R-tree [21], the GR-tree [5] and the 4R-tree [4].

The M-IVTT (Multiple Incremental Valid Time Trees) is a two level hierarchical index based on B^+-trees. In the upper level, one tree indexes the transaction time of events, having its leaves pointing to valid time trees. Underneath this transaction time tree there is a forest

[1]A spacestamp can be either a N-dimensional MBR (Minimum Bounding Rectangles), a point or any other appropriate abstraction.

of Valid Time Trees (VTTs.) Each VTT indexes the valid time of all records existing at that point in (transaction) time. Due to potentially large demand for space, only some VTTs are kept full, along with sufficient information (patches) on how to reconstruct any of the other ones.

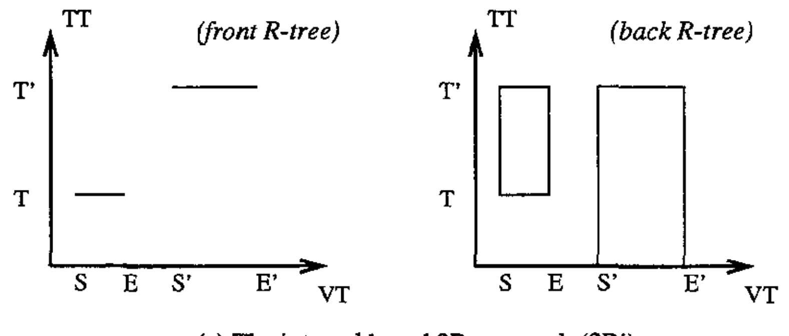
The Bitemporal Interval Tree (BIT), the Bitemporal R-tree (BRT) and 2R-tree structures index closed valid time ranges and now-relative transaction time objects. The BIT and BRT follows the partial-persistent methodology. In a partially persistent structure only the newest version of an object can be modified, whereas in an ephemeral structure old versions of an object are discarded when an update occurs. The authors modify an ephemeral memory based structure with good worst-case performance into the BIT, which is disk based, partially persistent and well paginated. The BRT makes an R*-tree [1] a partially persistent directed acyclic graph of pages. The structure is then formed by several logical R-trees, representing the evolution of objects in the transaction time sense.

The 2R-tree uses two R-trees (named *front* and *back* R-trees) to index bitemporal data. The bitemporal domain is mapped to a two-dimensional space (valid time $\times$ transaction time) as follows. An object with an unknown transaction end time is stored in the front R-tree as a line. Recall that in this approach the valid time ranges are bounded and, naturally, the transaction start time is always known. Once this object is updated, it is removed from the front R-tree and is inserted into the back R-tree as a rectangle. Figure 5.1(a) shows an example of this approach. The front R-tree indexes two objects, inserted at (transaction) time T and T' and which are still current in the database, i.e., bear an open transaction end time. The back R-tree, on the other hand, indexes two other objects which were current in the database during [T, T'] and [0, T'] respectively.

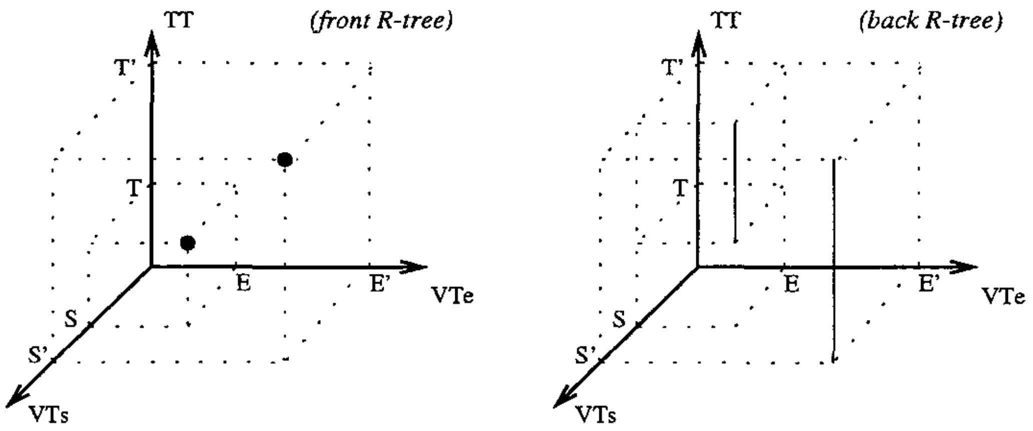
Note, however, the valid time interval can be transformed into a point in a three-dimensional space (valid start time $\times$ valid end time $\times$ transaction time). Likewise the rectangles formerly in the back R-tree are now transformed in three-dimensional segments. Figure 5.1(b) shows how the temporal data in Figure 5.1(a) would be indexed using such transformation. The advantage of using such a point based approach is that the amount of overlap among the indexed objects is diminished, hence the underlying R-trees can offer a better performance.

The GR-tree and 4R-tree index both now-relative valid and transaction time. The GR-tree extends the R*-tree [1] to store both static tuples (with closed valid and transaction time ranges) and growing objects (with either valid or transaction end time unknown). In this new tree, the indexed objects in its nodes can be either a growing rectangle or a growing stair-shape object, in addition to the standard MBRs supported by the R*-tree. By storing such growing objects, the dead space among objects in the GR-tree is decreased when compared to using the R*-tree and hence it becomes much more efficient.

To reduce dead space, the 4R-tree maps a growing rectangle into a closed line and a



(a) The interval based 2R approach (2Ri)



(b) The point based 2R approach (2Rp)

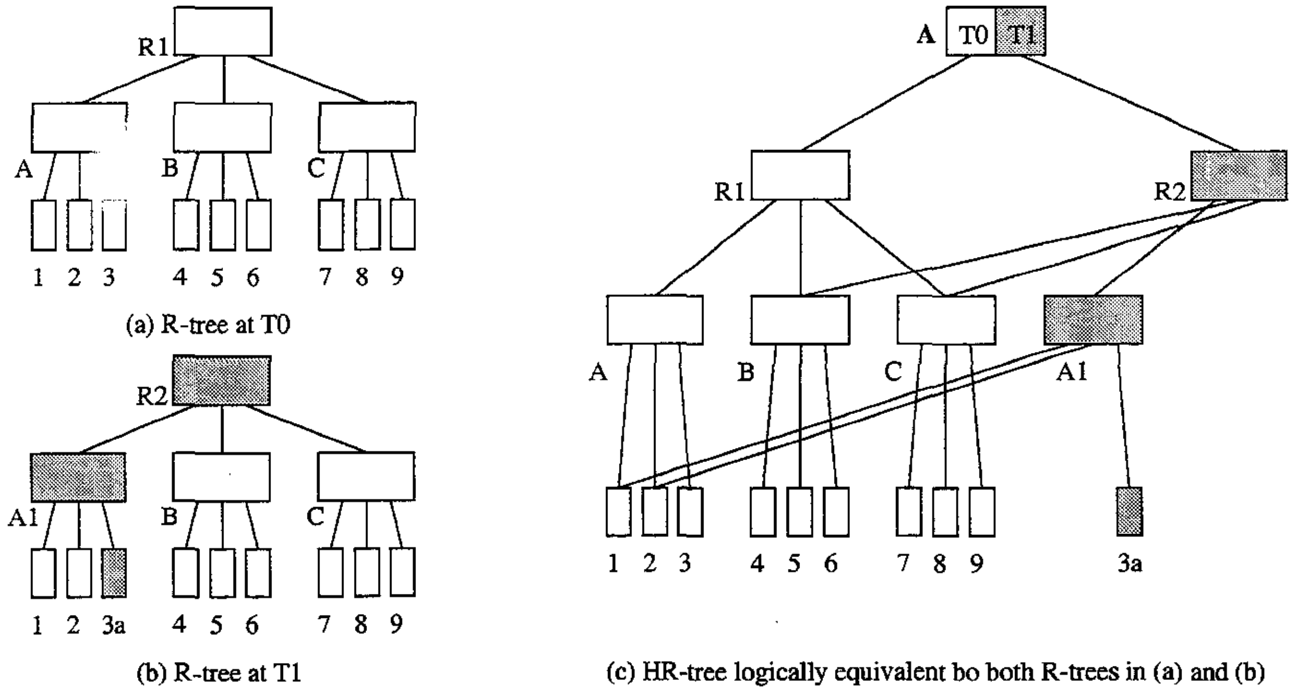
Figure 5.1: The two variants of the 2R-tree approach.

growing stair-shape object to a point. Using such a transformation the proposed approach is able to use "off-the-shelf" R-trees (which is the main goal of the proposal). Indeed, the objects are indexed in four R*-trees, depending on whether their valid and transaction end time are open or not. As objects are updated they may move between such R*-trees like in the 2R-tree approach. In fact, it is interesting to note that in the case of bitemporal data with no now-relative valid time the 4R-tree reduces to the 2R-tree.

Even though we have not discussed, in all approaches above incoming queries should be modified accordingly, further details can be found in the original papers.

5.3 The HR-tree for 2TBDs

As most other proposals the HR-tree is also based on R-tree. The HR-trees were designed as a spatiotemporal indexing structure, as such, let us use a spatiotemporal scenario as a motivation. Consider an object O which lies at spacestamp S_0 during time $[t_0, t_1)$ and then lies at spacestamp S_1 during $[t_1, t_2)$ and so on and so forth. These "movements" characterize different states (snapshots) of the spatiotemporal database. One trivial way to index such states would be to build an R-tree for each of them. Although this is obviously not a practical solution, it is reasonable to assume that sibling R-trees may have some (potentially many) identical nodes. The HR-tree explores this, by keeping all previous states of the two-dimensional R-tree *only logically* instead of physically. This is achieved by allowing consecutive instances of R-tree to overlap, i.e., to share nodes. This idea was also proposed in [25] but for B^+-trees in the context of (single dimension) temporal databases. As an illustration consider the two consecutive (with respect to their timestamps) R-trees in Figure 5.2(a) and (b), which can be represented in a much more compact manner as the HR-tree shown Figure 5.2(c). In this example all objects (at the leaf node level) but object 3 are current in the database as of time T1 and as such have their transaction end time open (i.e, equal to *now*). Object 3 on the other hand has its transaction time equal to [T0, T1), meaning that a query posed at transaction time T1 traverses the *logical* R-tree rooted at R2 and does not "see" object 3, as one would expect.



(a) R-tree at T0

(b) R-tree at T1

(c) HR-tree logically equivalent bo both R-trees in (a) and (b)

Figure 5.2: A single HR-tree logically equivalent to multiple R-trees.

Although it is just a simple example, it is easy to see that much space could be saved by re-using the nodes that did not change from a given state to the next one. Note that with the addition of a simple structure **A** (an array in the figure, but which could be a B-tree if warranted) the root node of the desired R-tree, current for a given timestamp, can be obtained quickly, and thus the query processing cost is the *same* as if all R-trees where kept physically. This becomes handy in the case of transaction time point queries. However, should one query a transaction time range, then several logical R-trees would need to be searched, which could be costly. Details about the HR-tree structure as well as its companion algorithms can be found in [27, 28].

As argued earlier in the paper it is rather simple to use the HR-tree to index 2TDBs with now-relative transaction time. Bounded valid time ranges can be considered degenerate two-dimensional MBRs, which the R-tree can handle well, and thus also the HR-tree. Given that, the overall idea is to: (1) index the initial set of tuples under an HR-tree; and (2) as tuples are updated new branches under the HR-tree are created.

Note that it is highly desirable to keep the number of newly created branches in the HR-tree as low as possible. For that reason some R-tree variants are not suitable to serve as HR-tree's framework, notably, the R*-tree [1]. That structure avoids node splitting by forcing re-insertion, which is likely to affect several branches, hence "swelling" the HR-tree. A similar reasoning also applies to the R^+-tree [34] which may duplicate entries. Among the other alternatives, we have found the Hilbert R-tree[2] [18] to be suitable for our purposes and use it as HR-tree's baseline. It does not yield duplication, avoids re-insertion and is indeed reported to be quite efficient.

As the BRT, BIT, 2R-tree, GR-tree and 4R-tree the HR-tree is based on R-trees. Unlike the GR-tree and 4R-tree, and like the BRT and BIT, the HR-tree was not designed to handle now-relative valid time. Differently than the 2R-tree and 4R-tree, the HR-tree maintains only one single structure. A unique feature of the HR-tree is that it is able to query each indexed database state (logical R-tree) as if it was stored individually. This provides a very good query processing time, with very little, if any, overhead, unlike all other structures. There is, however, the price of a potentially large overhead in space. We investigate this issue, among others, in the next section.

5.4 Performance Analysis

We now present some of the results obtained when investigating the HR-tree's performance. As the papers proposing the BRT, BIT, GR-tree and 4R-tree did, we will use the 2R-tree approach (described in Section 5.2), as references against which we compare our proposal – in fact, we will use both approaches, the interval based (which we refer to as 2Ri) and the point based (which we refer to as 2Rp). All R-trees used in the experiments, including the one used as a basis for the HR-tree, are Hilbert R-trees implemented as described in [18].

As usual in this area we focus on three main issues: update cost, query processing cost and storage requirements (the first two are measured in terms of disk I/Os). Before discussing

[2]Note that HR-tree should not be confused with a shorthand for Hilbert R-tree, in fact, HR-tree stands for Historical R-tree.

the figures obtained let us sketch how the data sets we used were generated. Some of the following criteria have been inspired by [21, 5].

We have used only data sets with closed valid time ranges, that is, the initial and end valid time are known. All the data sets have 100,000 updates (insertions or deletions). Each indexing structure is initially populated with 5,000 insertions, which is followed by 95,000 insertions/deletions. Three differents groups of data sets were generated, each one having different insertions/deletions ratios, namely: 60/40, 75/25 and 90/10. From now on we refer to these groups as the 60/40, 75/25 and 90/10 (data) files, respectively. Finally, each data group has four files, varying the number of updates per transaction timestamp, we experimented this number being 100, 500, 1,000 and 5,000. This reflects how better (or worse) a given structure handle different sizes of batch updates per transaction timestamp. Notice that this implies in data files having from 1000 to 20 transaction timestamps.

Without loss of generality all time values are real numbers between 0 and 1. This is due to the implementation of the Hilbert R-tree (thus the HR-tree) we currently have and is not a limitation of the structures presented. The average length of the valid time ranges is 0.05 (i.e., 5% of the maximum timespan) and it was generated using an exponential distribution.

5.4.1 Update Cost

The first issue investigated was the cost for updating the indexing structures. Figure 5.3 presents the average number of disk pages accessed per update in all three structures for the 60/40 data files. The HR-tree has the lowest average I/O per update, followed by the 2Ri and 2Rp. All structures benefit (though not considerably) from having a larger batch of updates per transaction timestamp. The HR-tree outperforms the 2R approach because at each transaction timestamp the logical R-tree updated in the HR-tree is smaller than the R-tree updated by the 2R approach. In the HR-tree just one logical R-tree is "visible" per transaction timestamp, whereas in the 2R approaches all updates regardless of their transaction timestamp are "visible" under the same structure. Therefore the HR-tree can be updated more efficiently. As for the two other data files (75/25 and 90/10) we noticed that as the insertions/deletions rate increases, the HR-tree requires slightly more I/Os, while both variations of the the 2R improve their performance. In fact, we also observed that for the 90/10 data file the 2Ri performs nearly as well as the HR-tree.

In general, the lower the insertions/deletions ratio, the better the HR-tree's relative performance. This was indeed verified in the remaining of the experiments. This can be explained as follows. The higher the number of deletions per transaction timestamp the higher the likelihood that nodes already modified in those transaction timestamps (by the deletions) can be re-used. Hence, new nodes need not be created, enhancing update time. When there is a much larger the number of insertions (relative to the number of deletions) there is a higher probability that new nodes need be created, hence consuming disk I/Os.

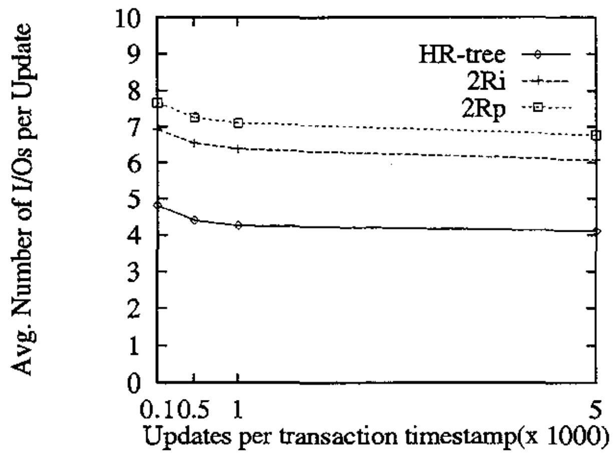


Figure 5.3: Cost of updates, 60/40 data file

5.4.2 Query Processing Cost

To query the data indexed, we have performed transaction time point and valid time point/range queries, denoted respectively as */point/point and */range/point queries, after simplifying the notation introduced in [45]. In both cases, the transaction time is randomly chosen from one of the transaction timestamp indexed. For the */point/point case the valid time is a random time point within [0, 1). For */range/point queries, the queried time length has also an exponential distribution with average equal to 0.05. Each query file created has 250 queries and the average figures are the ones reported.

As note earlier the HR-tree may not yield good performance when querying transaction time ranges, i.e., queries of the type: */point/range and */range/range. Indeed, this was verified in [28] in the context of spatiotemporal databases. As such, we chose not to deal with such type of queries in this paper.

Figure 5.4 presents the average number of disk pages accessed per query in point queries for the 60/40 data files. The HR-tree has the best query performance, requiring about 68% less disk access than the 2Rp, which was expected to be the best of the 2R based implementations. As discussed above, for the 75/25 and 90/10 data files, the HR-tree's loses some of its relative advantage. Nevertheless it still offers the best query performance, being about 50% and 33% faster than the 2Rp, respectively.

Figure 5.5 depicts the results for */range/point queries, using the 60/40 date file and both the indexed and query ranges with average lengths of 5% of the maximum timespan. Consistently, the HR-tree yield the best performance, being about 77% faster, than the 2Rp which again outperforms the 2Ri. The HR-tree remains the best struture when using the 75/25 and 90/10 data files, being at least 50% faster than the 2Rp.

We have also experimented smaller and large query ranges, with average length of 1% and

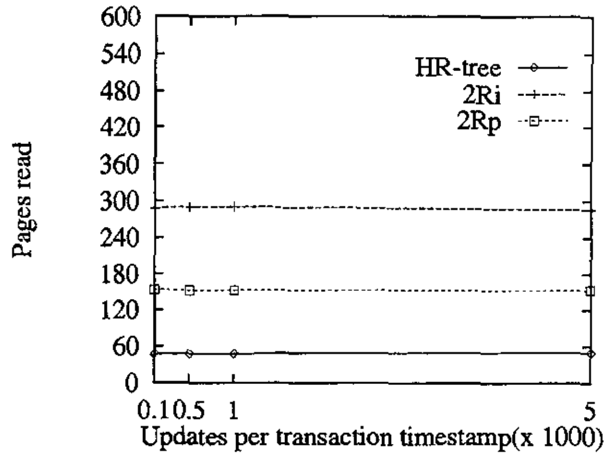


Figure 5.4: Cost of */point/point queries.

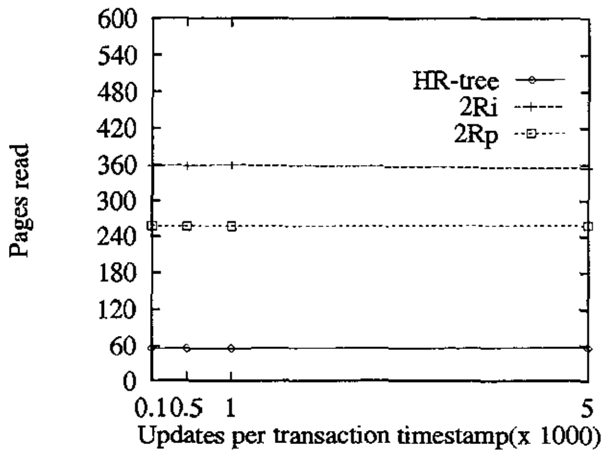


Figure 5.5: Cost of */range/point queries.

10% of the maximum timespan. For the 1% case, the performance was virtually the same as the one obtained in the */point/point case (Figure 5.4). When using larger queries we noticed that the relative advantage of the HR-tree becomes even larger. In fact, it becomes over 80% faster than the 2Rp and the 2Rp's curve becomes closer to the 2Ri's. While in Figure 5.5 the 2Rp is about 33% faster than the 2Ri, when querying larger ranges (twice as large in this case) this advantages falls to around 15%. When querying points (Figure 5.4 or short valid time ranges (the 1% case) the 2Rp was over 40% faster than the 2Ri. This shows that the gain obtained by indexing points instead of ranges (thus diminishing the degree of overlap) may be lost when querying large regions. This behavior was observed when using the 75/25 and 90/10 data sets as well, where again the HR-tree, which is always the faster index, loses its relative advantage as the ratio of insertions/deletions increases.

5.4.3 Storage Requirements

Figure 5.6 shows the size of the indexes created, for the 60/40 data file. The 2Ri and 2Rp structures have a linear behavior as the number of updates per transaction timestamp increases since the number of objects indexed, i.e., the number of updates, in the structure does not increase. On the other hand, the larger the number of updates per transaction timestamp the lower the size of HR-tree. As argued earlier, this is so because less tree branches are duplicated. This leads us to claim that the HR-tree is not suitable for scenarios with a low number of updates per transaction timestamp, even though the performance does not change nearly as much, and the HR-tree is consistently the faster index. The 2Ri implementation yields an index sligthly smaller than the 2Rp one. In the 2Ri, objects in the two dimensional space are stored, insted of the 2Rp, which stores three dimensional objects. This implies in more objects per page in the 2Ri against the 2Rp, which means a lower index structure. For the 75/25 and 90/10 data files the results were qualitatively similar. Quantitatively, however, the HR-tree's curve shifts up faster with the increase in the insertion/deletion rate.

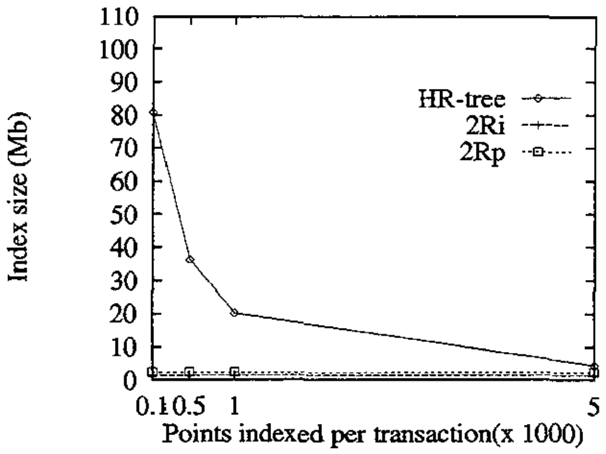


Figure 5.6: Indices sizes.

5.5 Conclusions

Due to some similarity between spatiotemporal and bitemporal objects, we have investigated the use of a spatiotemporal index structure, the HR-tree to the 2TDB indexing problem. Using the 2R-tree [21] as a reference, we focused mainly on indexing bounded (i.e., not now-relative) valid time ranges and now-relative transaction time. Performance was measured upon queries of type */point/point and */range/point. The structures query performance and sizes were investigate with respect to: number of updates per transaction timestamp;

insertions/deletions ratio; and size of the queried valid time ranges. Even though we do not deal with the indexing and querying of non-temporal data, e.g., keys, either the HR-tree and the 2R-trees (as well as all other R-tree based structures) could be used for such a task.

Regarding the above variables we noticed that as the number of updates per transaction timestamp increases, the HR-tree's size diminishes, indicating that HR-tree is better suited for batch updates. Likewise the lower the ratio insertion/deletion of objects the better it is for the HR-tree. Overall, the HR-tree has a good update performance. The only case where it nearly tied with the 2R approach happened when the number of insertions was much bigger than the number of deletions. For all data files and queries investigated, the HR-tree yielded consistently the best search performance, requiring, most of the time, less than half of the I/Os required by the best 2R-tree approach. Size-wise, the HR-tree is very dependent on the update rate. The more updates per transaction timestamp, the smaller the resulting tree.

Several applications may present such high update rate characteristic, e.g., in the banking/financial domain. With current technology one could have updates bearing very fine transaction timestamps, say milliseconds. However, it is hardly reasonable to consider that queries will be posed using such a fine granularity. As such all transactions happening within, say a minute, could bear the same transaction timestamp and be inserted into the index in a batch mode. We believe that this rationale would also apply to other application domains. Another possibility, requiring some small further processing at update time, would be to collect all incoming transactions along with their original (fine grain) timestamps in a buffer and index all the updates in that buffer at pre-specified time intervals. In such a case, if the user poses a query with respect to a lower level timestamp than the one actually indexed (say milliseconds instead of minutes) then false-hits would likely need to be filtered out of the query's answer. Notice that such filtering would avoid access to actual data records (i.e., useless I/Os). The user could then experiment with differents time granularities in order to decrease false-hits at the possible expense of obtaining a larger (but fast nevertheless) index.

As the BTR was also compared to the 2R approach, an indirect comparison between the HR-tree and BRT performance presented in [21] seems to indicate that the structures may have comparable search performance for the queries investigated in this paper. We should make clear though that the BRT experiments assume one single update per transaction timestamp. Such an update rate would render the HR-tree unfeasibly large. We only conjecture that the HR-tree may be comparable to the BRT in the case of reasonably sized batch updates. Even though the GR-tree and 4R-tree were also compared against the 2R-trees a direct comparison between those and the HR-tree cannot be easily made as those structures assume now-relative valid time, which is not the case of the HR-tree.

Future research should focus on: (*i*) investigating how the HR-tree performs when indexing now-relative valid time (perhaps after some modification in its structure); (*ii*) investigating the effect of cache structures; (*iii*) designing a benchmarking data set against which previously proposed structures could be evaluated and compared and; (*iv*) investigating whether the overlapping approach could be used with other range indexing structures (e.g., [3]).

Acknowledgments

Jefferson R. O. Silva was supported by FAPESP (Process 97/11205-8). Mario A. Nascimento was partially supported by CNPq (Process 300208/97-9) and is also a researcher with Embrapa (mario@cnptia.embrapa.br). This research was developed as part of MCT/PRONEX's project SAI. The authors thank S. Saltenis and C. S. Jensen for their constructive comments.

Capítulo 6

Conclusão

Essa dissertação teve como proposta principal a criação e avaliação de uma nova estrutura de acesso à dados espaço-temporais.

O capítulo 2 apresentou a estrutura proposta, além dos algorítmos básicos para inserção e remoção de dados, e consultas espaço-temporais.

O capítulo 3 investigou o problema de geração sintética de dados espaço-temporais, apresenta um algoritmo (GSTD) para geração de tais dados, dá exemplos de conjuntos de dados gerados com o algoritmo e apresenta as bases de uma arquitetura para um ambiente de benchmarking para estruturas de acesso à dados espaço-temporais.

No capítulo 4 apresentou-se o trabalho de comparação da HR-tree com a 2+3 R-tree e 3D R-tree. Os dados utilizados foram gerados sinteticamente, utilizando-se o algoritmo GSTD. Basicamente, as seguintes conclusões podem ser tiradas:

- A HR-tree apresenta uma performance muito superior às outras duas em respostas à consultas espaciais em um ponto específico no tempo,

- Para consultas espaciais em intervalos de tempo, a HR-tree é superior à 2+3 R-tree e 3D R-tree para intervalos de tempo pequenos,

- A 3D R-tree é a estrutura que utiliza menos I/Os na construção de índices, sendo seguida pela HR-tree e 2+3 R-tree,

- A HR-tree gera arquivos de índice muito maiores que a 2+3 R-tree e a 3D R-tree. Uma abordagem diferente foi utilizada na inserção dos dados de forma a reduzir o tamanho do índice gerado pela HR-tree. Verificou-se uma diminuição de até 45% no tamanho. Nessa abordagem, há um overhead no número de páginas de disco lidas pela HR-tree em consultas. No entanto, verificou-se que mesmo assim a HR-tree continua a ser a melhor estrutura em consultas espaciais em pontos específicos no tempo,

- Quanto mais estático o ambiente sendo indexado, ou seja, quanto menos atualizações por timestamp forem realizadas, menor o arquivo de índice gerado pela HR-tree.

O capítulo 5 aborda o problema de indexação de dados bitemporais. A HR-tree foi utilizada como estrutura de acesso à dados bitemporais e comparada com outra estrutura (2R-tree) proposta em [20]. As seguintes conclusões podem ser tiradas:

- A HR-tree apresenta uma performance muito superior à 2R-tree em respostas à consultas em um ponto específico no tempo,
- Para a construção do índice, a HR-tree também é superior à 2R-tree, acessando menos páginas de disco,
- A HR-tree gera arquivos de índices bem maiores que a 2R. Porém, adotando-se uma técnica de inserção em *batch*, foi possível diminuir o tamanho do índice gerado pela HR-tree e ainda assim manter sua superioridade em consultas.

De forma resumida, consideramos como contribuições da dissertação os seguintes pontos:

- Desenvolvimento, implementação e avaliação de uma nova estrutura de acesso à dados espaço-temporais, a HR-tree,
- Desenvolvimento e implementação de um gerador de dados espaço-temporais sintéticos, disponível via www[1]
- Testes de desempenho da HR-tree com duas outras estruturas,
- Aplicação da HR-tree em outro domínio de problemas, a indexação de dados bitemporais.

Alguns trabalhos podem ser feitos continuando o trabalho feito nessa dissertação. Consideramos como possíveis extensões os seguintes pontos: (i) novos testes da HR-tree, considerando outros tipos de dados, como MBRs ou dados mistos (MBRs e pontos); (ii) avaliar como *cache* afeta o desempenho da HR-tree; (iii) estudar políticas de "merging" e "purging"; (iv) criação de uma lista ligando versões de um mesmo objeto, de modo a permitir responder consultas sobre históricos de objetos; (v) criação do ambiente de benchmarking proposto no capítulo 3 - basicamente é a criação de um site que contenha (a) um módulo gerador de dados (GSTD), (b) um repositório de conjuntos de dados reais, (c) um conjunto de estruturas de acesso, (d) um banco de dados com resultados experimentais e (e) uma ferramente para visualização de dados gerados com o GSTD.

[1]http://www.dbnet.ece.ntua.gr/~theodor/GSTD/ e http://www.dcc.unicamp.br/~mario/GSTD/

Apêndice A

Algoritmos básicos de inserção, remoção, atualização e consultas na HR-tree

Os algoritmos apresentados no artigo do capítulo 2 são algoritmos que definem a HR-tree sobre uma R-tree [15]. Porém, como pode ser visto no artigo do capítulo 4, a HR-tree foi implementada utilizando-se a Hilbert R-tree[18] como estrutura espacial base.

Desse modo, esse capítulo apresenta os algoritmos necessarios para inserção, remoção, atualização e consultas de dados na HR-tree.

São permitidos atualizações na HR-tree em *batch*, ou seja, várias atualizações em um mesmo instante de tempo.

O algoritmo *AdjustTree* é o algoritmo original da Hilbert R-tree [18]. O algoritmo *HandleOverflow* é basicamente o algoritmo original da Hilbert R-tree, com uma pequena alteração: quando um nó irmão do nó que está recebendo a nova entrada é usado pelo algoritmo e sua marca de tempo é menor que *now*, então o nó deve ser duplicado.

A notação usada é apresentada na Tabela A.1.

```
Algorithm HR-Insert(On, HR)
1. { create a new state in the HR-tree }
   if A[pt] < now
      then create a new entry in A indexing now;
2. { create a root NR to insert On }
   invoke CreateBranch(On, HR) to create a new logical R-tree rooted at NR.
   The new logical R-tree contains a leaf node L in which to place On;
3. { insert On in L }
   if L has room for another entry
      then insert On in L;
```

Tabela A.1: Notation used in the algorithms.

HR	ponteiro para a HR-tree
A	o vetor de tempo da HR-tree
R	nó raiz de uma R-tree
On	MBR inserido na HR-tree
Oo	MBR removido da HR-tree
F	uma entrada em um nó da R-tree
p	ponteiro associado à F
N.Nt	marca de tempo do nó N
L,LP,N,NR	ponteiros para nós da R-tree
S	conjunto de nós da árvore
SW	um MBR representando uma janela de consulta
now	ponto corrente no tempo
pt	entrada mais recente em A
t	uma entrada de tempo indexada em A
it,et	início e fim de um intervalo de tempo

```
      else invoke HandleOverflow(L, On). A new leaf may be created if split
          was done.
4. { propagate split upwards }
   form a set S that contains L, its cooperating siblings and the new leaf
          node, if any.
   invoke AdjustTree(S).
5. { grow tree taller }
   if node split propagation caused a root split
      then create a new root NR whose children are the nodes resulting from
          the split;
   adjust the entry in A to point to the new root NR;

Algorithm HR-CreateBranch(On, HR)
1. { initialize }
   set N to be the root pointed to by A[pt] in HR;
   if N.Nt < now
      then create a new node L;
           copy all entries of N into L;
           set L.Nt = now;
           set NR = L;
      else set NR = N;
           set L = N;
2. { leaf check }
   if N is a leaf
      then return NR and L;
3. { choose subtree }
   let F be the entry with the minimum hilbert value grater than the hilbert
      value of On
4. { create a new node of the new branch and
     descending the tree }
   set LP = L;
   set N to be the node pointed to by F.p;
   if N.Nt < now
      then create a new node L;
           copy all entries of N into L;
           set L.Nt = now;
           adjust the pointer F.p in LP to point to L;
      else set L = N;
5. { Loop until a leaf is reached }
   repeat from step 2;

Algorithm HR-Delete(Oo, HR)
1. { find the leaf node containing Oo}
   set R to be the root pointed to by A[pt];
   perform an exact match search to find the leaf node L that contain Oo.
   duplicate the branch that contains L if necessary.
   set the timestamp of the nodes duplicated as Now;
   if L cannot be found
      then stop;
2. {create a new state in the HR-tree if necessary }
  if the root was duplicated
      create a new entry in A with time value equal to now;
      Remove Oo from L;
4. if L underflows
      then borrow some entries from s cooperating siblings.
           duplicate the siblings if their timestamp < now
   if all the siblings are ready to underflow
      then merge s+1 to s nodes.
           duplicate them if their timestamps < now
      adjust the resulting nodes.
5. { adjust tree }
   form a set S that contains L and its cooperating siblings (if underflow
      has occurred).
   invoke AdjustTree(S).
6. { shorten tree }
   if the root node has only one child after the tree has been adjusted
      then make the child the new root R;
   adjust A[now] to point to R;
```

Usando os algoritmos acima é fácil implementar o movimento de um ponto de sua localização corrente para uma nova posição em um dado instante de tempo. Basta remover o ponto de sua posição corrente e inserí-lo na nova posição. O correspondente algoritmo é apresentado a seguir:

```
Algorithm HR-Move(Oo, On, HR)
1. { Delete the current MBR version }
   Invoke Delete passing Oo and HR;
2. { Insert the new MBR version }
   Invoke Insert passing On and HR;
```

A seguir são apresentados dois algoritmos para consulta de dados espaciais (i) em um instante específico de tempo e (ii) em um intervalo de tempo, respectivamente.

```
Algorithm HR-SearchPoint(S, t, HR)
1. { find the appropriate root R }
   if t = now
      then set R to be the node pointed to by A[pt];
      else set R to be the node pointed to by A[t];
2. { find the MBRs which overlap S }
   invoke original Hilbert Search algorithm
     passing R;

Algorithm HR-SearchRange(S, it, et, HR)
1. { find the appropriate root R at time it (initial time) }
   if it = now
      then set R to be the node pointed to by A[pt];
      else set R to be the node pointed to by A[it];
2. { find the MBRs which overlap S }
   set t to be it
```

```
while (t <= et)
  invoke original Hilbert Search algorithm passing R;
  set t to be the next time in A[.];
  set R to be the node pointed to by A[t];
```

Referências Bibliográficas

[1] N. Beckmann, H. P. Kriegel, R. Schneider, and B. Seeger. The R*-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD International Conferencee on Management of Data*, pages 322–331, June 1990.

[2] D. Bitton, D. J. DeWitt, and C. Turbyfill. Benchmarking database systems: A systematic approach. In *Proceedings of the 9th Conference on Very Large Data Bases (VLDB)*, 1983.

[3] G. Blankenagel and R. H. Güting. External segment trees. *Algorithmica*, 12(6):490–532, 1994.

[4] R. Bliujūtė et al. Light-weight indexing of general bitemporal data. Technical Report 30, TimeCenter, 1998.

[5] R. Bliujūtė et al. R-tree based indexing of now-relative bitemporal data. In *Proc. of the 24th Intl. Conf. on Very Large Databases*, pages 345–356, August 1998.

[6] F. W. Burton et al. Implementation of overlapping B-trees for time and space efficient representation of collection of similar files. *The Computer Journal*, 33(3):279–280, 1990.

[7] F. W. Burton, M. W. Huntbach, and J. Kollias. Multiple generation text files using overlapping tree structures. *The Computer Journal*, 28(4):414–416, 1985.

[8] J. Clifford et al. On the semantics of "NOW" in temporal databases. *ACM Transaction on Database Systems*, 22(2):171–214, June 1997.

[9] Informix Corp. Developing datablade modules for informix dynamic server with universal data option. White paper, 1998.

[10] M. Erwig, R. H. Guting, M. Schneider, and M. Vazirgiannis. Abstract and discrete modeling of spatio-temporal data types. In *Proc. of the 6th ACM Intl. Workshop on Geographical Information Systems (ACM-GIS)*, 1998.

[11] V. Gaede and O. Günther. Multidimensional access methods. To appear in ACM Computing Surveys. `http://www.wiwi.hu-berlin.de/~gaede/survey.rev.ps.Z`, 1997.

[12] V. Gaede and O. Günther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):123–169, June 1998.

[13] J. Gray, P. Sundaresan, S. Englert, K. Backlawski, and P. Weinberger. Quickly generating billion-record synthetic databases. In *Proceedings of ACM SIGMOD Conference*, 1994.

[14] O. Gunther, V. Oria, P. Picouet, J. M. Saglio, and M. Scholl. Benchmarking spatial joins 'a la carte. In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management (SSDBM)*, 1998.

[15] A. Guttman. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the 1984 ACM SIGMOD International Conferencee on Management of Data*, pages 47–57, Jun 1984.

[16] Y. Ioannidis, M. Livny, S. Gupta, and N. Ponnekanti. Zoo: A desktop experiment management. In *Proceedings of the 22nd Conference on Very Large Data Bases (VLDB)*, 1996.

[17] C. S. Jensen, J. Clifford, S. K. Gadia, A. Segev, and R. Snodgrass. A consensus glossary of temporal database concepts. *ACM SIGMOD Record*, 23(1):52–64, Jan 1994.

[18] I. Kamel and C. Faloutsos. Hilbert R-tree: An improved R-tree using fractals. In *Proceedings of the 20th Very Large Databases Conference (VLDB'94)*, pages 500–509, Santiago, Chile, 1994.

[19] G. Kollios, D. Gunopulos, and V. J. Tsotras. On indexing mobile objects. Technical Report UCR-CS-98-06, University of California, Riverside, December 1998.

[20] A. Kumar, V. J. Tsotras, and C. Faloutsos. Access methods for bi-temporal databases. In *Proceedings of the International Workshop on Temporal Databases*, pages 235–254, 1995.

[21] A. Kumar, V. J. Tsotras, and C. Faloutsos. Designing access methods for bi-temporal databases. *IEEE Transactions on Knowledge and Data Engineering*, 10(1):1–20, 1998.

[22] S. Lanka and E. Mays. Fully persistent B$^+$-trees. In *Proceedings of the 1991 ACM SIGMOD International Conferencee on Management of Data (SIGMOD'91)*, pages 426–435, Denver, USA, May 1991.

[23] S. T. Leutenegger, M. A. Lopez, and J. Edgington. STR: An efficient and simple algorithm for R-tree packing. In *Proceedings of the 13th IEEE International Conference on Data Engineering (ICDE'97)*, pages 497–506, Birmingham, UK, April 1997.

[24] D. Lomet and B. Salzberg. Access methods for multiversion data. In *Proceedings of the ACM SIGMOD Conference*, pages 315–324, June 1989.

[25] Y. Manolopoulos and G. Kapetanakis. Overlapping B^+-trees for temporal data. In *Proceedings of the 5th Jerusalem Conference on Information Technology*, pages 491–498, August 1990.

[26] M. A. Nascimento, M. H. Dunham, and R. Elmasri. M-IVTT: An index for bitemporal databases. In *Proc. of the 7th Intl. Conf. on Databases and Expert Systems Applications*, pages 779–790, September 1996.

[27] M. A. Nascimento and J. R. O. Silva. Towards historical R-trees. In *Proc. of the 1998 ACM Symposium on Applied Computing*, pages 235 – 240, February 1998.

[28] M. A. Nascimento, J. R. O. Silva, and Y. Theodoridis. Access structures for moving points. Technical Report 33, TimeCenter, 1998.

[29] J. Orestein. Spatial query processing in an object-oriented database system. In *Proceedings of the 1986 ACM SIGMOD International Conferencee on Management of Data*, pages 326–336, May 1986.

[30] J. Patel et al. Building a scalable get-spatial dbms: Technology, implementation and evaluation. In *Proceedings of ACM SIGMOD Conference*, 1997.

[31] B. Salzberg and V. J. Tsotras. A comparison of access methods for temporal data. Technical Report 18, TimeCenter, 1997. *To appear in ACM Computing Surveys.*

[32] H. Samet. The quadtree and related hierarchical data structures. *ACM Computing Surveys*, 16(2):187–260, 1984.

[33] H. Samet. *The Design and Analysis of Spatial Data Strucutures.* Addison-Wesley, Reading, MA, 1990.

[34] T. Sellis, N. Roussopoulos, and C. Faloutsos. The R^+-tree: A dynamic index for multidimensional objects. In *Proceedings of the 13th Very Large Databases Conference*, pages 507–518, September 1987.

[35] A. P. Sistla, O. Wolfson, S. Chamberlain, and S. Dao. Modeling and querying moving objects. In *Proceedings. of the 13th IEEE Conference on Data Engeneering (ICDE)*, 1997.

[36] M. Stonebraker, J. Frew, and J. Dozier. The sequoia 2000 project. In *Proceedings of the 3rd International Symposium o Advances in Spatial Databases (SSD)*, 1993.

[37] M. Stonebraker, J. Frew, K. Gardels, and J. Meredith. The sequoia 2000 storage benchmark. In *Proceedings of ACM SIGMOD Conference*, 1993.

[38] A. Tansel et al., editors. *Temporal Databases: Theory, Design and Implementation.* Benjamin/Cummings, Redwood City, USA, 1993.

[39] Y. Theodoridis and T. Sellis. A model for the prediction of R-tree performance. In *Proc. of the ACM Symposium on Principles of Database Systems*, pages 161 – 171, June 1996.

[40] Y. Theodoridis, T. Sellis, A. Papadopoulos, and Y. Manolopoulos. Specifications for efficient indexing in spatiotemporal databases. In *Proceedings of the 10th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 123–132, July 1998.

[41] Y. Theodoridis, J. R. O. Silva, and M. A. Nascimento. On the generation of spatiotemporal datasets. Technical Report TR CH-99-01, Chorochronos, 1999.

[42] Y. Theodoridis, E. Stefanakis, and T. Sellis. Efficient cost models for spatial queries using R-trees. Technical Report 03, Chorochronos, 1998. To appear in *IEEE Transactions on Knowledge and Data Engineering.*

[43] Y. Theodoridis, M. Vazirgiannis, and T. Sellis. Spatio-temporal indexing for large multimedia applications. In *Proceedings of the 3rd IEEE Conference on Multimedia Computing and Systems (ICMCS)*, pages 441 – 448, June 1996.

[44] N. Tryfona. Modeling phenomena in spatiotemporal applications: Desiderata and solutions. In *Proc. of the 9th International Conference on Database and Expert Systems Applications (DEXA)*, 1998.

[45] V. J. Tsotras, C. S. Jensen, and R. T. Snodgrass. An extensible notation for spatiotemporal index queries. *ACM SIGMOD Record*, 27(1):47–53, 1998.

[46] V. J. Tsotras and A. Kumar. Temporal database bibliography update. *ACM SIGMOD Record*, 25(1):41–51, 1996.

[47] T. Tzouramanis, M. Vassilakopoulos, and Y. Manolopoulos. Overlapping linear quadtrees: a spatio-temporal access method. In *Proc. of the 6th ACM Intl. Workshop on Geographical Information Systems (ACM-GIS)*, November 1998.

[48] O. Wolfson, B. Xu, S. Chamberlain, and L. Jiang. Moving objects databases: Issues and solutions. In *Proc. of the 10th International Conference on Scientific and Statistical Database Management (SSDBM)*, pages 111 – 122, July 1998.

[49] M. F. Worboys. A unified model for spatial and temporal information. *The Computer Journal*, 1(37):26–34, 1994.

[50] X. Xu, J. Han, and W. Lu. RT-tree: An improved R-tree index structure for spatiotemporal databases. In *Proceedings of the 4th International Symposium on Spatial Data Handling (SDH)*, pages 1040 – 1049, 1990.

[51] C. Zaniolo et al., editors. *Advanced Databases Systems.* Morgan Kauffman, San Francisco, USA, 1997.

[52] J. Zobel, A. Moffat, and K. Ramamohanarao. Guidelines for presentation and comparison of indexing techniques. *ACM SIGMOD Record*, 25(3):10–15, 1996.