Partie C

Généralisation des mécanismes aux structures arborescentes

Les objets de grande taille se trouvent naturellement dans les bases de données ; il peut s'agir de champs longs ou de collections de grande cardinalité. Les versions de collection peuvent être représentées de différentes manières, en fonction du type relationnel ou orienté-objet des données gérées. On peut, dans le cas relationnel, voir une relation comme une collection de *n-uplets*. Cependant, peu de modèles [Abd97] gèrent des versions de relation et donc des versions de collection de *n-uplets*. Les modèles temporels [TCG+93] proposent un mécanisme de versionnement de données sous forme de versions historiques de données. Les versions historiques de données représentent un cas particulier de données temporelles, elles sont associées à une ou plusieurs dimensions du temps et ne peuvent pas être mises à jour ni être supprimées. Lorsque la dimension temporelle correspond au temps de transaction, c'est-à-dire au temps où un fait a été inséré dans la base de données [JCE+94], les données représentent l'historique de l'activité de la base de données plutôt que l'historique du monde réel modélisé [KTF98]. En revanche, lorsque la dimension temporelle correspond au temps de validité, c'est-à-dire au temps où un fait est vrai dans le monde réel modélisé [JCE+94], et que la base est en ajout seule (*append-only database*) [EWK93], les données correspondent à l'historique du monde modélisé.

Les bases de données orientées-objet gèrent des collections d'objets; il s'agit d'ensembles, de listes ou de regroupement d'objets de différents types (*bags*). Dans les SGBDOO, les collections sont par exemple employées comme points d'entrée dans la base (ou racines de persistance) [Cat97]. Dans les bases de données traditionnelles, lorsqu'une collection contient un nombre important d'éléments, c'est-à-dire lorsque sa taille est supérieure à celle d'une page de disque ou de la mémoire, elle est souvent organisée en index. Dans le SGBD O_2 [BDK92] par exemple, un ensemble est organisé par un index en arbre B [BM72], dans lequel les clés sont les identificateurs correspondant aux éléments de l'ensemble, ou les éléments eux-mêmes s'ils sont atomiques. Les collections d'objet peuvent également être vues comme des objets complexes, c'est-à-dire des objets constituants une entité composée de sous-entités, ses éléments. Les objets complexes sont par nature de longueur variable et peuvent dépasser la taille d'une page [DLR91].

Quelle que soit la manière de gérer une collection, dès que sa cardinalité est grande, les index sont indispensables pour optimiser l'accès à ses éléments [BO99].

La majorité des structures d'index dérivent ou combinent les propriétés des arbres B ou des arbres R. L'arbre V [MCD94], par exemple, inspiré de l'arbre B+ positionnel du système EXODUS [CDRS89] (voir chapitre 1), est considéré comme une variante de l'arbre R. Il est utilisé pour indexer les longues séquences de points dans un espace à deux dimensions. L'arbre B K-D (*K-D-B Tree*) [Rob81] combine les propriétés de l'arbre K-D remanié [BF79] et celles de l'arbre B, pour indexer des points multidimensionnels [Gün98]. L'arbre G (ou *G-tree* pour *Grid tree*) [Kum84] allie les avantages des arbres B et ceux des structures en grille (*Grid File*) [YM98b]. De même, l'arbre *Buddy* (*Buddy-Tree*) [SK90] peut être vu comme un compromis entre un arbre R et une structure en grille. En outre, une architecture permettant de construire tout type de structure d'accès pour les bases de données a été proposée [HNP95, Aok98]. Cette architecture, nommée GiST (*Generalized Search Tree*), unit à cet effet les caractéristiques de l'arbre B+ [Com79], la variante plus implantée de l'arbre B, et celles de l'arbre R.

Cette partie présente une généralisation des mécanismes, proposés dans la partie B pour gérer

des images similaires organisées en arbre quaternaire, aux structures arborescentes en général et en particulier aux structures basées sur les arbres B+ et les arbres R. L'organisation de cette partie est la suivante. Le chapitre 7 présente les structures d'Arbre B+ et d'Arbre R Génériques. Le chapitre 8 présente un état de l'art des versions de collection et des index permettant de gérer ces versions et compare les Arbres B+ et R Génériques aux autres approches.

Chapitre 7

Arbres B+ et R Génériques

Résumé

Les Arbres B+ et R Génériques correspondent à une généralisation des mécanismes de l'Arbre Quaternaire Générique aux arbres B+ et aux arbres R. Ce chapitre rappelle les caractéristiques des arbres B+ et R et définit les concepts propres aux structures génériques d'arbres B+ et R. Ces concepts sont les *états* d'arbre, la *Hiérarchie d'Arbres* et le concept de *racine générique*. Les algorithmes de recherche et d'insertion dans ces structures sont peu différents des algorithmes de recherche et d'insertion dans un arbre B+ et dans un arbre R classiques. Les Arbres B+ et R Génériques complètent l'édifice édifié par les différents travaux de l'approche des Versions de Base de Données en ajoutant la gestion d'index et en facilitant la gestion de versions de collection d'objets. Un Arbre B+ Générique gère des versions de collections d'objets. Un Arbre R Générique gère des versions de collection d'objets spatiaux.

Introduction

L'Arbre Quaternaire Générique, présenté dans le chapitre 5, a été proposé pour gérer des images similaires. L'objectif de ce chapitre est d'adapter les mécanismes de l'Arbre Quaternaire Générique aux arbres B+ et R donnant ainsi naissance à des Arbres B+ et R Génériques. Un Arbre B+ Générique permet d'organiser des objets volumineux, comme dans [CDRS89], ou des versions de collection de données (objets ou n-uplets) comme dans [Rei97]. Il peut également faciliter l'interrogation de versions dans le cas relationnel, ce type de versions ayant été proposé dans [Abd97]. Un Arbre R Générique permet d'étendre le domaine d'application de l'approche proposée dans [JMR99a]. En effet, l'arbre R est souvent utilisé dans le cadre de l'interrogation des images par le contenu [PF95, NPT97, CF98] ou lors de l'interrogation de données spatio-temporelles [SVP+96].

L'organisation de ce chapitre est la suivante. La section 7.1 rappelle la définition des arbres B+ et des arbres R. La section 7.2 présente les structures d'Arbre B+ et d'Arbre R Génériques. La section 7.3 décrit les algorithmes de ces structures. Finalement, la section 7.4 présente plusieurs domaines d'application de ces structures.

7.1 Structures d'index arborescentes équilibrées

L'arbre B+ et l'arbre R sont des structures classiquement utilisées et connues dans le domaine des bases de données. Leurs mécanismes et leurs algorithmes sont succinctement rappelés dans cette section, afin de positionner la proposition des Arbres B+ et R Génériques dans leur contexte.

Un index est une structure qui organise les données d'une base afin d'effectuer des recherches rapides et sélectives dans un large volume de données [BO99]. Cette structure permet de localiser les emplacements (mémoire ou disque) où sont stockées les données en associant une valeur de clé d'accès à un ou plusieurs enregistrements [EN89]. Les arbres équilibrés, comme les arbres B [BM72] ou les arbres R [Gut84], dérivés de l'arbre B [Sam89], sont des exemples de structures efficaces pour gérer de larges collections de données [BO99]. L'arbre B, qui nécessite un ordre total sur les valeurs de clés [SVP+96], est principalement utilisé pour organiser des données dont les attributs peuvent être ordonnés. En revanche, l'arbre R est utilisé pour organiser des données multidimensionnelles, comme les données spatiales.

Cette section présente les structures d'arbre B et d'arbre R. Elle est organisée comme suit. La sous-section 7.1.1 définit la structure d'arbre B, ainsi que sa variante la plus utilisée, l'arbre B+. La sous-section 7.1.2 décrit la structure d'arbre R.

7.1.1 Arbre B et variantes d'Arbre B

Un arbre B [BM72], appelé également *Btree* ou arbre de Bayer, est une structure d'accès équilibrée à une dimension [GG98]. Une définition détaillée de cette structure est donnée en a). En b), l'arbre B+ de [Com79], variante de l'arbre B, est décrit. Les algorithmes de cette structure sont rapidement présentés en c).

a) Arbre B

Un arbre B d'ordre m est un arbre qui satisfait les propriétés suivantes [Knu73] :

- Chaque nœud (non terminal) possède au maximum m fils.
- Chaque nœud (non terminal), excepté la racine, possède au minimum $\lceil m/2 \rceil$ fils, $\lceil m/2 \rceil$ étant égale à la partie entière de $(m/2)$.
- La racine, lorsqu'elle est un nœud non terminal, possède au moins deux fils (sinon il s'agit d'une feuille).
- Toutes les feuilles (nœuds terminaux) apparaissent au même niveau.
- Un nœud non terminal contenant k fils contient $(k-1)$ clés.

La figure 7.1 représente la structure générale d'un nœud interne (ou non terminal) d'arbre B d'ordre m. Cette figure est reprise de [EN89].

Chaque nœud interne est de la forme $< P_1, < K_1, Pr_1 >, < K_2, Pr_2 >, ..., < K_{q-1}, Pr_{q-1} >, P_q >$, avec q compris entre la partie entière de $(m/2)$ et m. Les clés d'accès sont ordonnées : $K_1 < K_2 < K_3 < ... < K_{q-2} < K_{q-1}$. Chaque pointeur P_i est un pointeur vers un sous-arbre B. Chaque pointeur Pr_i est un pointeur vers des données, c'est-à-dire vers un bloc du fichier de données contenant un enregistrement dont la valeur de clé est K_i. Pour chaque clé X d'un sous-arbre pointé par P_i, on a $K_{i-1} < X < K_i$ pour $i \in]1, q[$, $X < K_i$ pour $i = 1$ et $K_{i-1} < X$ pour $i = q$.

La hiérarchie de l'arbre grossit par la racine de sorte que tous les chemins de la racine aux feuilles ont même longueur [BJP98]. La figure 7.2 donne un exemple d'arbre B d'ordre 3.

b) Arbre B+

La structure d'arbre B permet des accès séquentiels triés aux données, puisque les clés sont triées dans l'arborescence [BJP98]. Cependant, un parcours séquentiel trié de l'arbre B, après lecture de toutes les clés d'une feuille (ou d'un nœud), nécessite de remonter au nœud père pour accéder à la feuille sœur (ou au nœud frère) et de continuer la lecture dans l'ordre trié. Une optimisation de la structure d'arbre B, pour les parcours séquentiels triés, consiste à dupliquer toutes les clés stockées dans les nœuds non feuilles au niveau des feuilles, puis à chaîner les feuilles entre elles dans l'ordre du tri [BJP98]. Ce type d'arbre est appelé arbre B+ [Com79].

Un nœud feuille (ou terminal) d'arbre B+ est de la forme $<< K_1, Pr_1 >, < K_2, Pr_2 >, ..., < K_{q-1}, Pr_{q-1} >, P_{next} >$, avec q compris entre la partie entière de $(m/2)$ et m. Le pointeur P_{next} pointe la feuille suivante. Comme précédemment (voir section a)), Pr_i est un pointeur vers les données de clés K_i. Chaque nœud interne est de la forme $< P_1, K_1, P_2, K_2, ..., P_{q-1}, K_{q-1}, P_q >$, avec $K_1 < K_2 < ... < K_{q-1}$. Le pointeur P_i pointe vers un nœud fils dont les clés X sont telles que $K_{i-1} \leq X < K_i$ lorsque $i \in]1, q[$, $X < K_i$ lorsque $i = 1$ et $K_{i-1} \leq X$ lorsque $i = q$ [EN89].

La figure 7.3 représente la structure des nœuds (internes et feuilles) d'un arbre B+ d'ordre m. La figure 7.4 donne un exemple d'arbre B+ d'ordre 3.

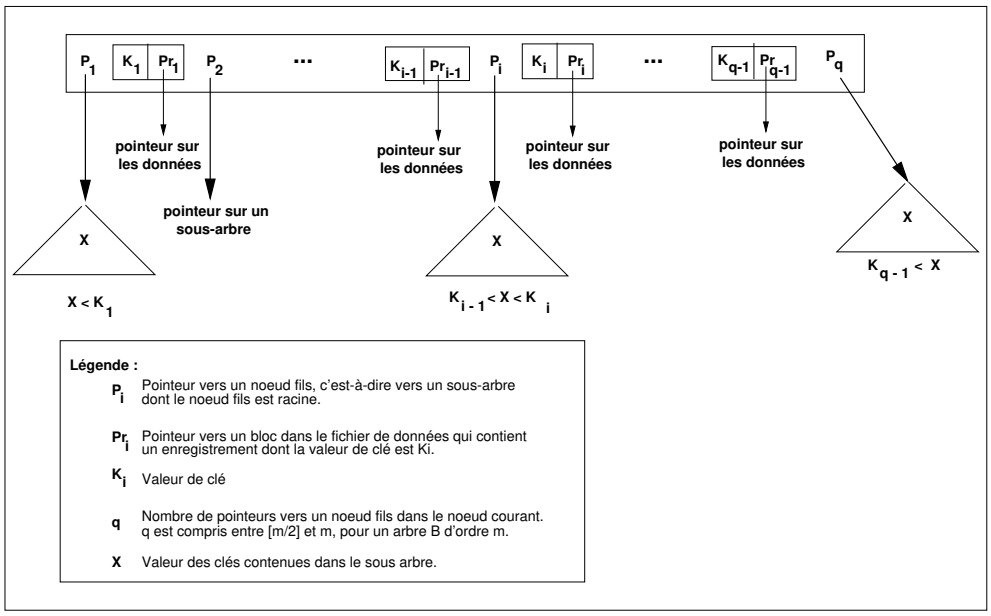

FIG. 7.1 – **La structure générale d'un nœud interne d'arbre B d'ordre** m.

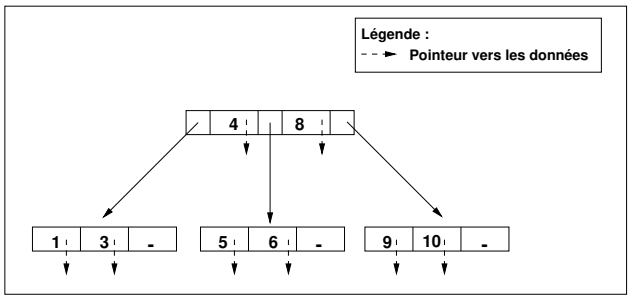

FIG. 7.2 – **Un arbre B d'ordre 3**.

L'arbre B+ est la variante de l'arbre B la plus implantée dans les bases de données [EN89]. La sous-section qui suit rappelle succinctement les algorithmes de cette structure.

c) Recherche et insertion dans un arbre B+

La recherche d'une clé dans un arbre B+ nécessite une recherche dichotomique dans chaque nœud de l'arborescence et ce, jusqu'à obtenir le nœud (ou la feuille) contenant cette clé. L'insertion d'une clé dans un arbre B+ peut entraîner l'éclatement des nœuds jusqu'à la racine. La suppression, en revanche, peut entraîner des fusions de nœuds et diminuer le nombre de niveaux de l'arbre. On parle de structure arborescente dynamique [GG98, BJP98].

Les algorithmes de recherche et d'insertion dans un arbre B+ sont donnés dans l'annexe A2. Pour plus de détails sur les algorithmes propres à l'arbre B, le lecteur est invité à consulter

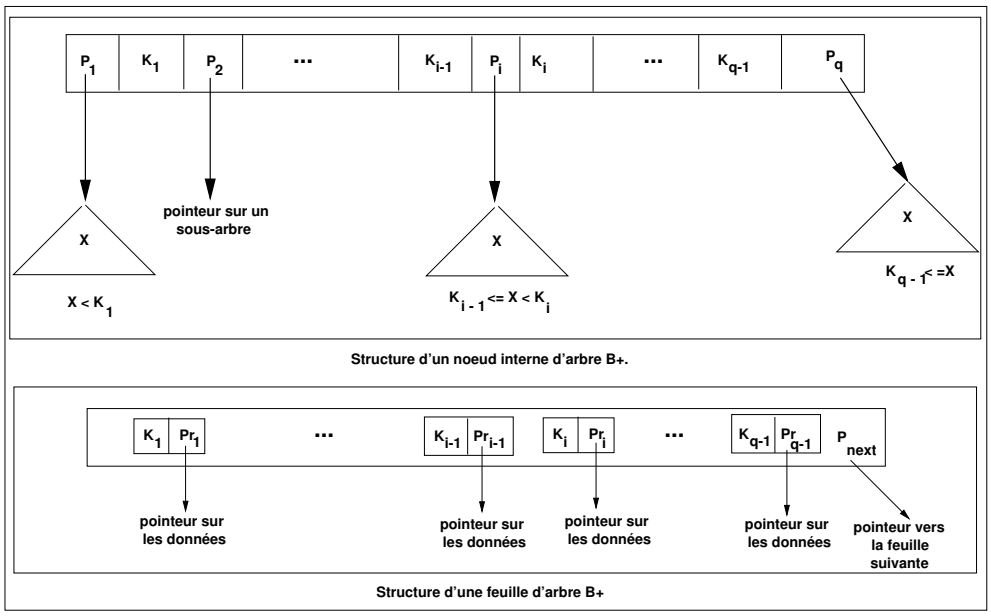

FIG. 7.3 – **La structure générale des nœuds d'arbre B+ d'ordre** m.

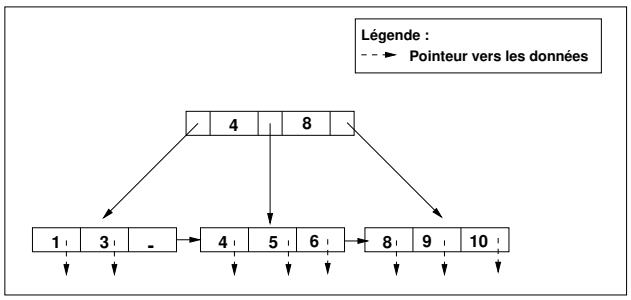

FIG. 7.4 – **L'arbre B+ correspondant à l'arbre B de la figure 7.2.**

[BM72, Knu73].

Exemple 7.1 La figure 7.5 représente un arbre B+ avant (à gauche de la figure) et après (à droite de la figure) l'insertion de l'enregistrement de clé 2. Suite à ces insertions, la structure de l'arbre n'est pas modifiée. En revanche, la figure 7.6 représente un arbre B+ avant (à gauche de la figure) et après (à droite de la figure) l'insertion d'un enregistrement de clé 11, où l'insertion modifie la structure de l'arbre.

L'insertion de l'enregistrement de clé 2 dans l'arbre B+ N°1 (situé à gauche de la figure 7.5) est effectuée comme suit. Le chemin menant de la racine à la feuille où insérer l'enregistrement de clé 2 est (n_0, f_0). Comme f_0 n'est pas pleine, l'enregistrement peut-être inséré dans la feuille f_0 (voir feuille f_0 en pointillé à droite de la figure 7.5).

Le chemin menant de la racine à la feuille où insérer l'enregistrement de clé 11, dans l'arbre

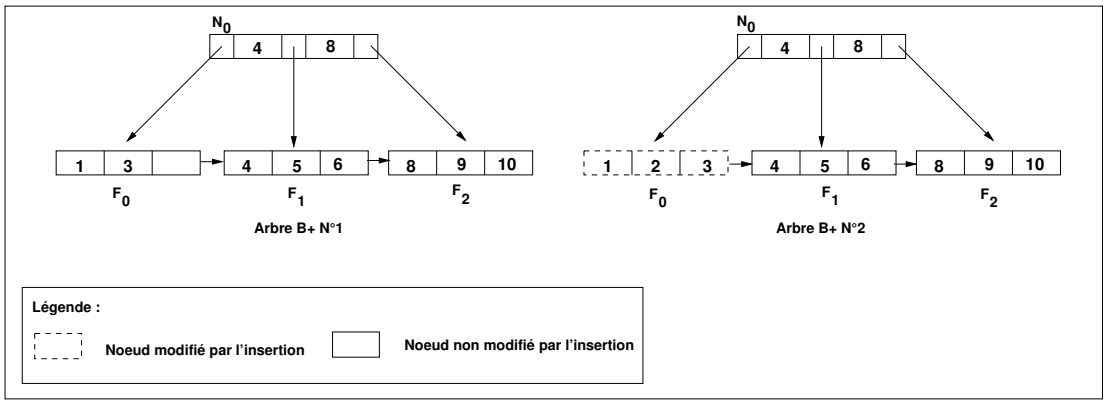

FIG. 7.5 – **Un arbre B+ avant et après l'insertion de l'enregistrement de clé** 2.

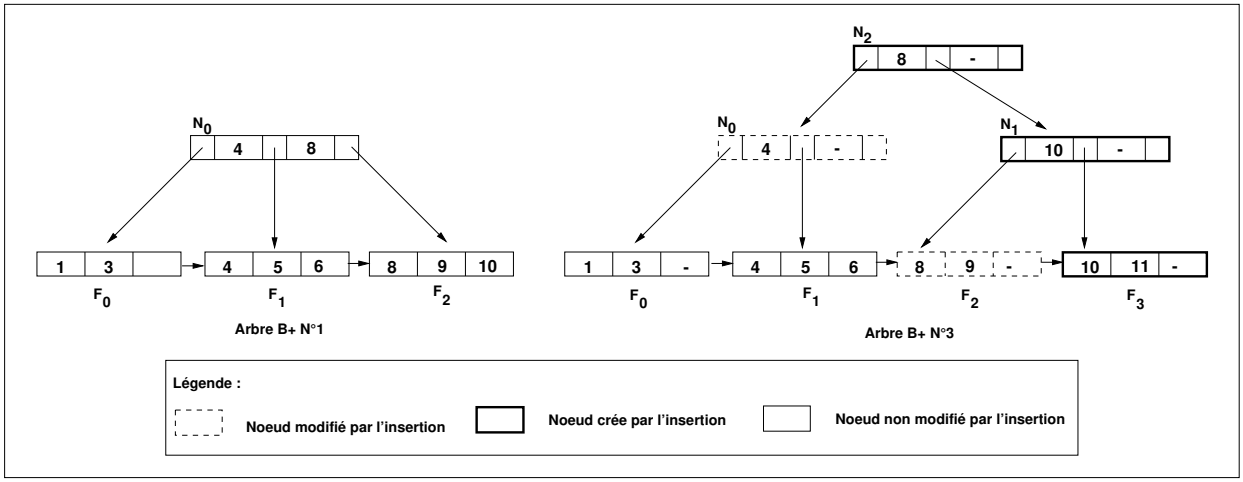

FIG. 7.6 – **Un arbre B+ avant et après l'insertion de l'enregistrement de clé** 11.

B+ situé à gauche de la figure 7.6, est (n_0, f_2). Or f_2 et sa sœur gauche f_1 sont pleines. Il faut donc créer une nouvelle feuille f_3 et répartir les clés de f_2 auxquelles est ajoutée la clé 11 entre les feuilles f_2 et f_3 (voir à droite de la figure 7.6). Puis, la feuille f_3 doit être ajoutée dans l'arbre B+, comme fille de n_0. Or le nœud n_0 est plein. Il faut, par conséquent, créer un nouveau nœud n_1 et répartir les feuilles f_0 à f_3 entre ces nœuds. Finalement, une nouvelle racine, n_2 doit être créée. Ses nœuds fils sont n_0 et n_1.

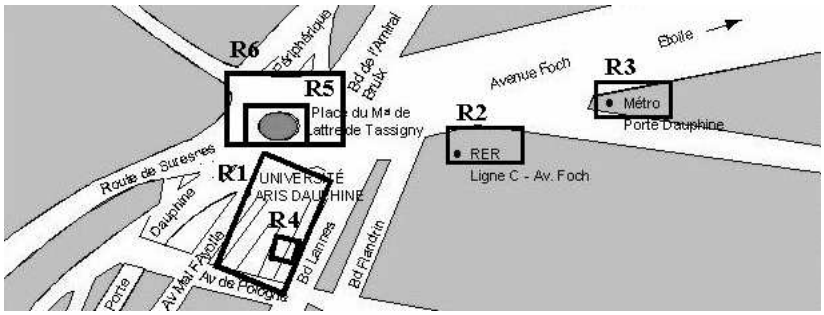

FIG. 7.7 – **La représentation de régions rectangulaires sur un fond de carte.**

7.1.2 Arbre R

L'arbre R [Gut84] (ou *R-Tree* en anglais) est une structure hiérarchique, dérivée de l'arbre B et utilisée pour indexer les objets géométriques ou spatiaux [SVP+96]. Les objets spatiaux sont représentés par des rectangles englobants minimums [1] sur une image ou un fond de carte [Sub98]. Par exemple, la figure 7.7 représente, sur un fond de carte, l'Université Paris Dauphine, la station de RER Foch, celle du métro Porte Dauphine, le bureau P626, la Place du Maréchal de Lattre de Tassigny, et le monument de cette place. Ces objets sont respectivement représentés sur le fond de carte par les rectangles englobants $R1$ à $R6$.

La sous-section a) décrit rapidement les caractéristiques de l'arbre R. La sous-section b) présente les algorithmes de cette structure.

a) Définition d'un arbre R

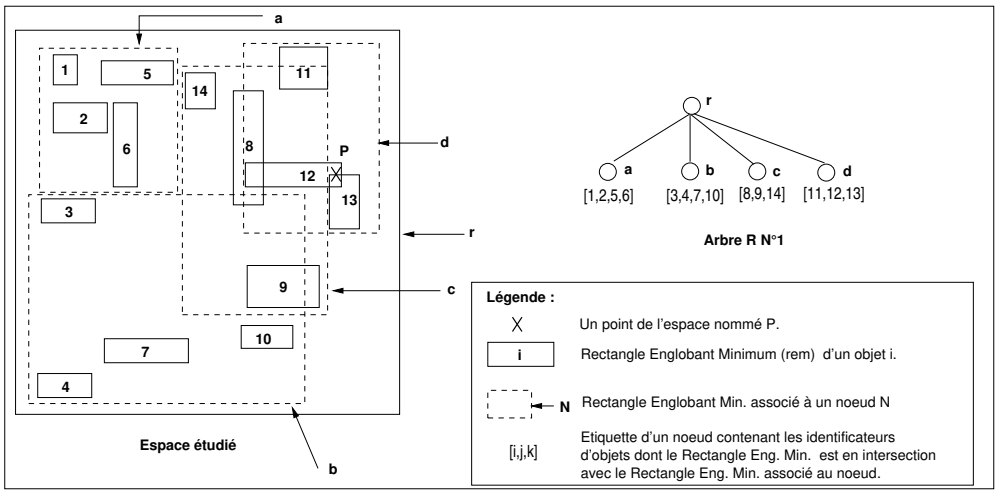

FIG. 7.8 – **Un arbre R.**

1. Le terme rectangle englobant minimum est noté, par la suite, *rem*. Il signifie *Minimum Bounding Rectangle* en anglais.

L'arbre R est utilisée pour organiser les données spatiales [Sam89, BJP98, YM98b] ou, plus précisément, les rectangles englobants minimums de ces données [SVP+96]. L'arbre R, comme l'arbre B, est un arbre équilibré de degré (ou ordre) m. Il présente les caractéristiques suivantes [SVP+96] :

- Un nœud interne d'arbre R permet d'adresser les nœuds du niveau suivant de l'arborescence.
- Un nœud feuille référence un ensemble d'objets spatiaux.
- Un nœud (interne ou feuille) est associé à une partie rectangulaire de l'espace de sorte que :
  - Le rectangle associé à un nœud feuille est le rectangle englobant minimum des rectangles englobants minimums des objets spatiaux référencés par le nœud feuille.
  - Le rectangle associé à un nœud intermédiaire est le rectangle englobant minimum des rectangles englobants minimums associés à chacun de ces fils.
  - La racine de l'arborescence est associée au rectangle englobant minimum de l'espace tout entier.
- Chaque nœud contient des entrées qui sont des couples (rem, oid) tels que :
  - Pour les nœuds internes : rem correspond au rectangle englobant minimum associé à un nœud fils, dont l'adresse est oid.
  - Pour les nœuds feuilles : rem est le rectangle englobant minimum d'un objet spatial dont l'adresse est oid.
- Chaque nœud, à l'exception de la racine, contient un nombre d'entrées m compris entre n et s, où n est le nombre minimal d'entrées par nœud et s et le nombre maximal d'entrées, tels que $0 \leq n \leq \lceil \frac{s}{2} \rceil$.
- La racine de l'arbre R possède au moins deux fils.

Exemple 7.2 La figure 7.8, reprise de [SVP+96], représente un arbre R (situé à droite de la figure), de paramètres $n = 2$ et $s = 4$, construit à partir d'une collection de quatorze objets (située à gauche de la figure). L'espace couvert par chacun des nœuds feuilles est représenté par un rectangle en pointillé. Il y a chevauchement des rectangles associés aux nœuds feuilles. Les rectangles englobants minimums contenus dans les nœuds d'arbre R ne sont pas représentés sur l'arbre R de la figure.

Il existe deux différences majeures entre un arbre B et un arbre R [YM98b]. La première différence est que les cellules de chaque nœud d'arbre B sont ordonnées, alors qu'elles ne le sont pas dans un nœud d'arbre R. La deuxième différence est qu'il n'y a pas de chevauchement entre les intervalles de clés dans un arbre B, alors que les rectangles englobants minimums peuvent se chevaucher dans un arbre R. Ces différences ont des impacts importants sur la manière de maintenir un arbre R et de faire des recherches via cet index [YM98b].

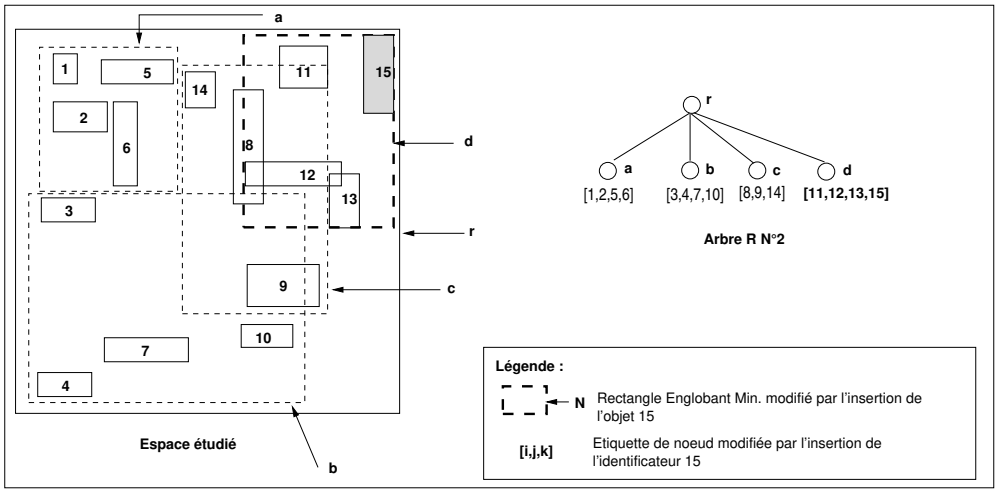

FIG. 7.9 – **L'arbre R de la figure 7.8 après insertion de l'objet** 15.

b) Algorithmes de recherche et d'insertion

L'arbre R permet de faire des requêtes d'intervalles ou de voisinage (*range query*) [RKV95, CF98], des requêtes sur les régions spatiales (*region query*) ou de requêtes particulières comme le *pointé* [SVP+96]. L'opération de pointé correspond à la désignation d'un point sur la carte ou à l'écran. Il s'agit d'une opération classique dans les bases de données géographiques qui consiste à rechercher les objets dont la géométrie contient un point donné. L'utilisateur désigne un point et demande au système d'exécuter par exemple la requête "*Dans quelle commune ou quel département est situé ce point?*".

Dans l'arbre R, le pointé consiste à parcourir l'arbre de la racine vers les feuilles contenant le point de l'espace recherché. Comme les rectangles se chevauchent, il est possible que la recherche impose un parcours de plusieurs chemins. L'algorithme du pointé dans un arbre R est donné en annexe A2.

L'insertion dans un arbre R est similaire à l'insertion dans un arbre B [SVP+96]. Toutefois, l'ordre d'insertion des objets dans l'arbre R a une influence déterminante sur la structure de l'arbre.

Exemple 7.3 Dans la figure 7.8, un point de l'espace, nommé P est indiqué par une croix. Un pointé sur P s'effectue de la manière suivante. (1) La racine r est lue afin de trouver dans quel rem des nœuds fils de r le point P est inclus. (2) P est inclus dans le rem de c ; il faut rechercher si le point P est inclus dans un des rem des objets référencés par c. (3) De même P est inclus dans le rem de d, il faut rechercher si le point P est inclus dans un des rem des objets référencés par d. On déduit que le point P est inclus dans les rem des objets 12 et 13.

Exemple 7.4 Les figures 7.9 et 7.10 représente l'arbre R de la figure 7.8 après insertion des objets 15 (grisé sur la figure 7.9) et 16 (grisé sur la figure 7.10). Ces figures sont reprises de [SVP+96]. La première présente une insertion qui ne modifie pas la structure de l'arbre R. La deuxième insertion en revanche modifie la structure de l'arbre dont le nombre de niveaux augmentent.

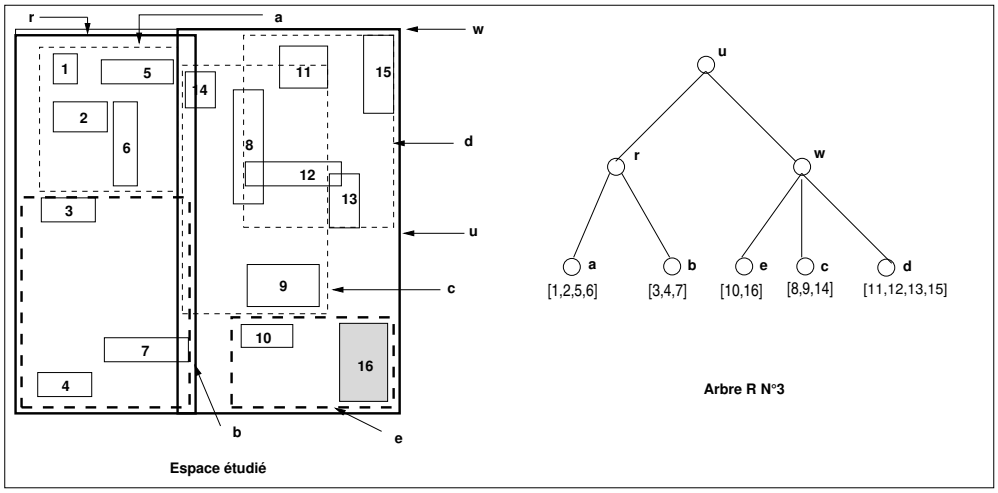

FIG. 7.10 – **L'arbre R de la figure 7.9 après insertion de l'objet** 16.

L'objet 15 (voir figure 7.9) peut être inséré dans le rectangle englobant d, après agrandissement de celui-ci (voir pointillés en gras). Le nœud feuille d n'étant pas plein, l'objet 15 y est inséré. L'objet 16 (voir figure 7.9) peut être inséré dans la feuille b, mais celle-ci est saturée (elle contient déjà quatre rem). Elle est par conséquent divisée en deux et donne naissance à une feuille e telle que b contienne les objets $3, 4$ et 10 et e contienne les objets 10 et 16. Le nœud r qui possède quatre fils est à son tour divisé, donnant naissance au nœud w. Une nouvelle racine u est ajoutée à l'arbre R.

7.2 Arbres B+ et R Génériques

L'accroissement de nouvelles applications telles que les bases de données spatio-temporelles ou les bases de données multimédia impose la création de nouveaux index ou l'adaptation de structures existantes [BO99]. La section précédente a présenté deux types d'index très utilisés, les arbres B et les arbres R. Plusieurs adaptations de l'arbre B aux structures évolutives (données temporelles ou versions de données) ont été proposées dans la littérature [CDRS89, GS93, ST94, BGO+96, VV97] (voir chapitre 8). De même, l'arbre R a été étendu pour permettre la gestion de données temporelles [KS91, Kol93, BJSS98, NS98] ou de données similaires [KS97], comme les images [PF95, PF97] ou les données multimédias en général [CF98]. Ces approches de la littérature sont comparées, dans le chapitre 8, avec les structures proposées dans cette section.

Nous proposons dans ce chapitre deux structures : l'*Arbre B+ Générique* et l'*Arbre R Générique*. L'Arbre B+ Générique permet de gérer des versions d'objet de grande taille ou des versions de collection d'objets de grande cardinalité organisées en arbre B+. L'Arbre R Générique permet de gérer des versions de collection d'objets spatiaux indexée par un arbre R.

Cette section présente le vocabulaire employé dans la suite du document pour présenter les structures génériques d'arbre B+ et d'arbre R. La sous-section 7.2.1 définit le concept d'*états* d'arbre B+ ou d'arbre R. La sous-section 7.2.2 décrit le concept de *Hiérarchie d'Arbres*. La sous-section

7.2.3 définit les *nœuds génériques* en général, en présentant un nouveau concept qui est celui de *racine générique*.

7.2.1 Etats d'arbre

Les *états* (ou versions) d'un arbre B+ ou d'un arbre R correspondent aux différentes modifications subies par l'arbre après plusieurs insertions ou suppressions. Chaque état de l'arbre est identifié.

Exemple 7.5 Les arbres B+ des figures 7.5 et 7.6 correspondent à trois états d'un même arbre B+. De même, les arbres R des figures 7.8, 7.9 et 7.10 correspondent aux différents états d'un même arbre R.

7.2.2 Hiérarchie d'Arbres

Une *Hiérarchie d'Arbres* correspond à l'organisation de différents états d'un même arbre. Un état dérive d'un autre parce qu'il y a eu une mise à jour par rapport à l'état précédent. L'organisation des arbres en Hiérarchie d'Arbres se fait en fonction des mises à jour effectuées sur les arbres. L'utilisateur peut choisir de faire directement des mises à jour sur un arbre, ou de créer un nouvel état reflétant les mises à jour effectuées sur l'arbre. On parle de Hiérarchie d'Arbres B+ et de Hiérarchie d'Arbres R.

Exemple 7.6 Les Arbres B+ des figures 7.5 et 7.6 sont organisées en hiérarchie telle que l'arbre B+ N°1 est l'arbre père des arbres B+ N°2 et N°3. La Hiérarchie des Arbres B+ est représentée en haut à droite de la figure 7.11. L'arbre B+ N°2 (situé à droite de la figure 7.5) dérive de l'arbre B+ N°1 suite à l'insertion de l'enregistrement de clé 2 dans l'arbre N°1. L'arbre B+ N°3 (situé à droite de la figure 7.6) reflète l'insertion de l'enregistrement de clé 11 dans l'arbre B+ N°1.

7.2.3 Nœuds et racine génériques

Un Arbre Générique, qu'il s'agisse d'un arbre B+ ou d'un arbre R, est composé de *nœuds génériques*. Comme dans l'Arbre Quaternaire Générique [JMR99a], un nœud générique d'Arbre B+ Générique (resp. d'Arbre R Générique) est composé de lignes. Chaque ligne contient une liste d'identificateurs d'arbre B+ (resp. d'arbre R) et une valeur de nœud d'arbre B+ (resp. d'arbre R). La valeur d'un nœud d'arbre B+ est une liste de couples *(clé, identificateur)*. La valeur d'un nœud d'arbre R est une liste de couples *(rem, identificateur)*.

Toutefois, la racine de ces structures est considérée comme un nœud générique particulier. En effet, à l'opposé de l'arbre quaternaire [Sam84], les arbres B+ ou R croissent par la racine. Par conséquent, en fonction des insertions ou des suppressions dans l'arbre, la racine de l'arbre peut changer. Dans le cas de la structure générique, le terme de *racine générique* est donc employé afin de permettre la gestion de l'évolution de la racine de l'arbre, c'est-à-dire le suivi des différentes racines dans les différents états de l'arbre.

Une racine générique est composée de lignes. Chaque ligne l contient une liste d'identificateurs d'arbre B+ ou R, notée $(a_1, ..., a_p)$, et l'identificateur d'un nœud générique n_g. Une ligne $[(a_1, ..., a_p)$,

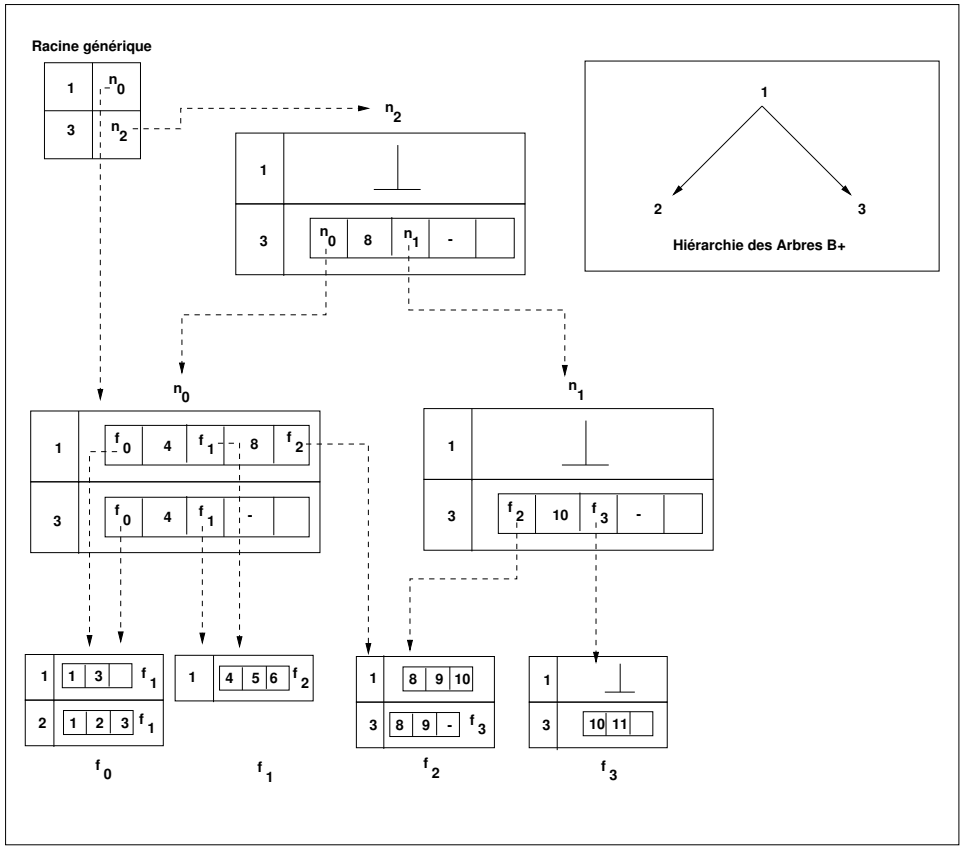

FIG. 7.11 – **L'Arbre B+ Générique contenant les arbres B+ des figures 7.5 et 7.6.**

n_g] de racine générique signifie que le nœud générique n_g est racine de l'arbre dans les états identifiés par a_i, $i \in [1, p]$. La valeur de cette racine est accessible à la lecture du nœud générique n_g. La valeur de ce nœud peut varier d'un état à l'autre de l'arbre.

Exemple 7.7 La racine générique de l'Arbre B+ Générique de la figure 7.11 (situé en haut à gauche de la figure) contient deux lignes. La première ligne signifie que la racine de l'arbre B+ N°1 est n_0. La deuxième ligne indique que le nœud n_2 est racine dans l'arbre B+ N°3.

La gestion des nœuds génériques décrits ci-avant est la même que celle des nœuds génériques de l'Arbre Quaternaire Générique. Le partage entre arbres peut être implicite : "*Excepté lorsque l'identificateur d'un arbre, a_i, est associé à une autre valeur v, l'arbre a_i partage implicitement la valeur associée à son arbre père.*". Toutefois, la valeur d'un nœud générique diffère en fonction du type de l'arbre.

a) Nœuds génériques d'Arbre B+ Générique

L'objectif de la structure d'Arbre B+ Générique est de permettre le suivi de l'évolution d'un arbre B+. Chaque nœud (interne ou feuille) est identifié. Les différents états d'un arbre B+

peuvent partager des nœuds de même identificateur et de même valeur.

Exemple 7.8 Dans la figure 7.5, les deux arbres B+ peuvent partager les nœuds non modifiés par l'insertion de l'élément de clé 2, c'est-à-dire les nœuds n_0, f_1 et f_2. De même, dans la figure 7.6, les deux arbres B+ peuvent partager les nœuds non modifiés par l'insertion de l'élément de clé 11, c'est-à-dire f_0 et f_1.

Un nœud générique n d'Arbre B+ Générique est composé de lignes. Chaque ligne contient une liste $(a_1, ..., a_p)$ d'identificateurs d'arbre B+ et une valeur v. Lorsque le nœud est interne, la valeur v est de la forme $< N_1, K_1, P_2, K_2, ..., N_{q-1}, K_{q-1}, N_q >$. La valeur de q est la même que celle indiquée dans la section b). Les pointeurs P_i vers les nœuds fils des nœuds d'arbre B+ classique sont remplacés par les identificateurs des nœuds génériques fils, notés N_i. Les identificateurs a_i, $i \in [1, p]$, d'arbre B+ associés à une valeur v dans une ligne de nœud générique n signifient que le nœud n a pour valeur v dans les arbres B+ identifiés par a_i. Lorsque la valeur est $\perp$, cela signifie que le nœud n'existe pas dans les arbres B+ associés. Lorsque le nœud est feuille, sa valeur est de la forme $< K_1, ..., K_p, N_{Next} >$, avec N_{Next}, l'identificateur de la feuille suivante.

Exemple 7.9 La valeur $< f_0, 4, f_1, 8, f_2 >$ du nœud générique n_0 (voir figure 7.11) signifie que, dans l'arbre B+ N°1 : les enregistrements de clé inférieure à 4 sont accessibles par la feuille f_0, les enregistrements de clé comprise entre 4 et 8 sont accessibles par la feuille f_1 et les enregistrements de clé supérieure à 8 sont accessibles par la feuille f_2. Comme l'identificateur de l'Arbre B+ N°2 n'apparaît pas dans le nœud générique n_0, on en déduit, par partage implicite, que la valeur du nœud n_0 dans l'arbre B+ N°2 est la même que celle qui est associée à son ancêtre dans la Hiérarchie d'Arbres, l'arbre B+ N°1. Par conséquent, la valeur du nœud n_0 est $< f_0, 4, f_1, 8, f_2 >$ dans l'arbre B+ N°2.
De même, la valeur $< 1, 3, - >$ du nœud générique f_0 est associée à l'arbre B+ N°1. Le nœud feuille f_0 contient donc les enregistrements de clé 1 et 3 dans l'arbre B+ N°1. En revanche, une autre valeur de f_0 est associée à l'arbre B+ N°2, dans le nœud générique f_0. On déduit que les enregistrements de clé 1, 2 et 3 sont éléments de l'arbre B+ N°2.

b) Nœuds génériques d'Arbre R Générique

Comme pour l'Arbre B+ Générique, la structure d'Arbre R Générique permet le suivi de l'évolution d'un arbre R. Les différents états d'un arbre R peuvent partager des nœuds de même identificateur et de même valeur.

Exemple 7.10 L'arbre R de la figure 7.8 peut partager tous les nœuds, excepté les nœuds identifiés par r et d, avec l'arbre R de la figure 7.9. En effet, dans l'arbre R de la figure 7.9, le nombre d'éléments référencés par d passe de trois à quatre et le rectangle englobant associé à d dans le nœud r est modifié. De même, l'arbre de la figure 7.9 peut partager les nœuds a et c avec l'arbre R de la figure 7.10.

Un nœud générique n d'Arbre R Générique est composé de lignes. Chaque ligne contient une liste $(a_1, ..., a_p)$ d'identificateurs d'arbre R et une valeur v. Lorsque le nœud est interne, la valeur v est une liste de couples (rem_i, n_i) où n_i est un nœud fils de n de rectangle englobant minimum rem_i inclus dans le rectangle englobant associé à n. Lorsque le nœud est feuille, la valeur v est une liste

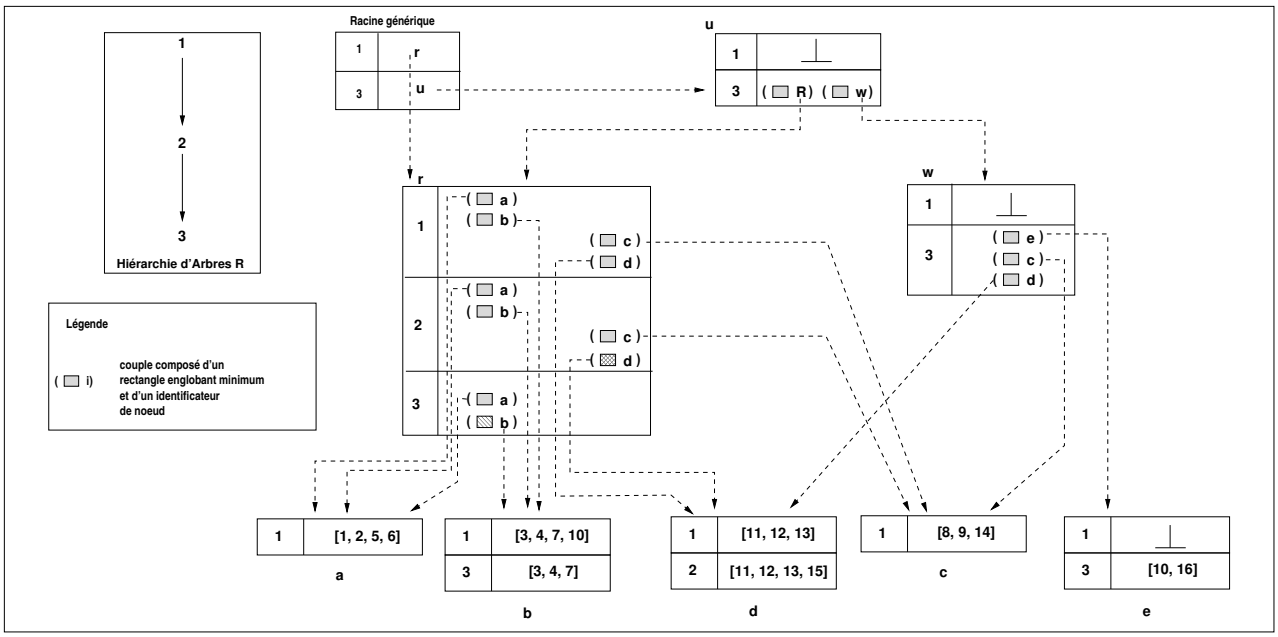

FIG. 7.12 – **L'Arbre R Générique contenant les arbres R des figures 7.8 à 7.10.**

de couples (rem_i, o_i) où o_i est un objet spatial de rectangle englobant minimum rem_i inclus dans le rectangle englobant de n.

Exemple 7.11 La figure 7.12 représente un Arbre R Générique stockant trois arbres R identifiés de 1 à 3. La racine de l'Arbre R N°1 est le nœud r. La racine de l'arbre R N°3 est le nœud w. Par partage implicite, on déduit que la racine de l'arbre R N°2 est la même que celle de l'arbre R N°1, c'est-à-dire le nœud r. Les rectangles englobants minimums associés aux nœuds sont représentés sur la figure par des petits rectangles grisés. Les rectangles englobants des objets référencés dans les nœuds feuilles ne sont pas représentés sur la figure.

Le nœud r a trois valeurs, une pour chaque arbre R. Ses fils, dans l'arbre R N°1, sont les nœuds a, b, c et d. Ses nœuds fils sont également a, b, c et d, dans l'arbre R N°2, mais le rectangle englobant du nœud d a été modifié par l'insertion de l'objet 15 (le rectangle englobant associé à d est quadrillé sur la figure). Seuls a et b sont fils du nœud r dans l'arbre R N°3.

Le nœud générique d contient deux lignes. La première ligne signifie que le rectangle englobant minimum de d inclut les rectangles englobants minimums des objets identifiés par 11, 12 et 13, dans l'espace représenté par l'arbre R N°1. La deuxième ligne signifie que le rectangle englobant minimum de d inclut en plus l'objet 15 dans l'espace représenté par l'arbre R N°2. On déduit qu'il en est de même, par partage implicite entre les arbres R N°2 et 3, dans l'espace représenté par l'arbre R N°3.

La section qui suit présente les algorithmes de recherche et d'insertion dans un Arbre B+ Générique et dans un Arbre R Générique.

7.3 Algorithmes de recherche et d'insertion

Les algorithmes de recherche et d'insertion dans un Arbre B+ Générique sont présentés en 7.3.1. Ceux de l'Arbre R Générique sont présentés en 7.3.2.

7.3.1 Rechercher et insérer dans un Arbre B+ Générique

a) Recherche d'un enregistrement de clé K dans un arbre B+ a_i

La recherche d'un enregistrement de clé K dans un arbre B+ a_i stocké dans un Arbre B+ Générique diffère peu de la recherche d'un enregistrement de clé K dans un arbre B+ classique. L'algorithme de recherche dans un arbre B+ classique est rappelé en annexe A2. Dans un Arbre B+ Générique, il faut ajouter les deux opérations suivantes :

1. La racine de l'arbre B+ a_i doit être recherchée : il faut pour cela rechercher à quelle valeur de la racine générique est associé l'arbre B+ a_i.

2. Chaque lecture d'un nœud n est précédée par la recherche, dans le nœud générique n, de la valeur de n dans l'arbre B+ a_i.

Exemple 7.12 La recherche de l'enregistrement de clé 2 dans l'arbre B+ N°2, stocké dans l'Arbre B+ Générique de la figure 7.11 (voir page 116), est réalisée de la manière suivante :

1. La racine l'arbre B+ N°2 est recherchée : l'identificateur 2 n'apparaît pas dans la racine générique de l'Arbre B+ Générique. Par partage implicite, comme l'arbre B+ N°2 est fils de l'arbre B+ N°1 dans la Hiérarchie d'Arbres, on déduit que la racine est la même dans les deux arbres. Par conséquent, le nœud n_0 est racine de l'arbre B+ N°2.

2. Par le même principe de partage implicite on déduit que la valeur de n_0 associée à l'arbre B+ N°2 est $(< f_0, 4, f_1, 8, f_2 >)$. Comme 2 est plus petit que 4, on déduit que l'enregistrement de clé 2 est accessible par la feuille f_0 dans l'Arbre B+ N°2.

3. La valeur du nœud générique f_0 associée à l'arbre B+ N°2 est $(1, 2, 3)$. L'enregistrement de clé 2 appartient bien à l'arbre B+ N°2.

b) Insérer un enregistrement de clé K dans un arbre B+ a_i

L'insertion d'un enregistrement de clé K dans un arbre B+ a_i peut se faire de deux manières :

1. L'utilisateur souhaite que l'insertion entraîne la création d'un nouvel arbre B+ dérivé de a_i dans la Hiérarchie d'Arbre B+ (voir exemple 7.13), pour ne pas perdre l'arbre a_i.

2. L'utilisateur souhaite que l'insertion se fasse par mise à jour de l'arbre B+ a_i sans création de nouvel arbre B+ (voir exemple 7.14).

Les seules différences entre l'algorithme d'insertion dans un arbre B+ stocké dans un Arbre B+ Générique et l'algorithme d'insertion dans un arbre B+ classique sont les suivantes :

1. La modification de la valeur v d'un nœud n par ajout, insertion ou répartition de clés dans l'arbre B+ a_i entraîne :

   - La création d'une nouvelle ligne dans le nœud n si la valeur v n'existe pas. Cette ligne associe la valeur v à l'arbre B+ a_i (voir exemples 7.13 à 7.15).
   - L'association explicite de l'arbre B+ a_i à la valeur v du nœud générique n, si cette valeur existe (voir exemple 7.13).
   - La suppression du partage implicite entre l'arbre B+ a_i et ses descendants dans la Hiérarchie d'Arbres dont l'identificateur n'est associé à aucune autre valeur v' de n. Cette suppression se fait par ajout des identificateurs d'arbres B+, fils de a_i dans la Hiérarchie d'Arbres, à l'ancienne valeur de n dans l'arbre a_i (voir exemple 7.15).

2. Si l'insertion entraîne l'éclatement d'un nœud, c'est-à-dire la création d'un nouveau nœud dans l'arbre B+ a_i, deux cas se présentent :

   - Soit le nœud n a un nœud frère droit n_f dans l'arbre B+ parent de l'arbre a_i, dans la Hiérarchie d'Arbres, tel que la valeur de n_f dans l'arbre a_i soit $\perp$. Dans ce cas, la répartition des clés de n se fait entre n et n_f, modifiant la valeur de n_f dans l'arbre a_i (voir exemple 7.15).
   - Soit le nœud n n'a pas de nœud frère droit n_f dans l'arbre B+ parent de l'arbre a_i, dans la Hiérarchie d'Arbres, tel que la valeur de n_f dans l'arbre a_i soit $\perp$. Dans ce cas, la création d'un nouveau nœud new dans l'arbre B+ a_i entraîne la création d'un nœud générique new dans l'Arbre B+ Générique. Ce nœud contient dans un premier temps une seule ligne associant la valeur $\perp$ à l'arbre B+ racine de la Hiérarchie d'Arbres. Cette association indique que le nœud new n'existe dans aucun arbre B+. La valeur de new dans l'arbre B+ a_i est, ensuite, ajoutée au nœud générique new. De plus, il faut supprimer le partage implicite de la valeur de new entre l'arbre B+ a_i et ses arbres descendants dans la Hiérarchie d'Arbres (voir exemple 7.14) : les identificateurs des arbres fils de a_i sont explicitement associés à la valeur $\perp$ du nœud générique new.

Exemple 7.13 L'utilisateur souhaite insérer dans l'arbre B+ N°3 des enregistrements de clé 2 et 15. Il crée nouvel arbre B N°4 identique à l'arbre B N°3, puis le met à jour par insertion des clés 2 et 15, car il ne souhaite pas perdre l'arbre B+ N°3.

Dans un premier temps, l'arbre B+ N°4 est ajouté dans la Hiérarchie d'Arbres comme fils de l'arbre B+ N°3 (voir figure 7.13). L'arbre B+ Générique reste inchangé par cette insertion dans la Hiérarchie d'Arbre. Dans un deuxième temps, les enregistrements de clé 2 et 15 sont insérés dans l'arbre B+ N°4.
L'insertion de l'enregistrement de clé 2 est réalisé comme suit. L'algorithme de recherche permet de connaître le chemin de la racine à la feuille où l'enregistrement de clé 2 doit être inséré dans l'arbre B+ N°4 : (n_2, n_0, f_0). La feuille f_0 n'est pas pleine dans l'arbre B+ N°4, car la valeur de f_0

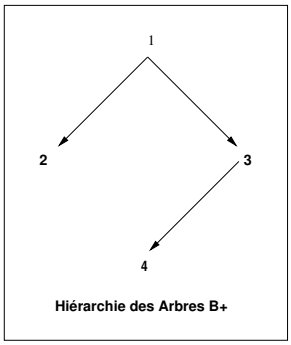

FIG. 7.13 – **La Hiérarchie d'Arbres B+ après la création d'un nouvel arbre B+ N°4 de même valeur que l'arbre B+ N°1.**

associée implicitement à l'arbre B+ N°4 est $(1, 3, -)$. La valeur de la feuille f_0, après insertion, est $(1, 2, 3)$. Or cette valeur existe dans le nœud générique f_0 et la feuille suivante de f_0, dans l'arbre B+ N°4, est f_1. L'insertion se traduit par conséquent par l'ajout de l'identificateur de l'arbre B+ N°4 dans la ligne de valeur $(1, 2, 3)$ du nœud générique f_0 (voir figure 7.14).
L'insertion de l'enregistrement de clé 15 est réalisé comme suit. L'algorithme de recherche permet de connaître le chemin de la racine à la feuille où l'enregistrement de clé 15 doit être inséré dans l'arbre B+ N°4 : (n_2, n_1, f_3). La feuille f_3 n'est pas pleine dans l'arbre B+ N°4. La clé 15 est ajoutée dans cette feuille : une nouvelle ligne est ajoutée au nœud générique f_3, associant l'arbre B+ N°4 à la valeur $(10, 11, 15)$.

Exemple 7.14 L'utilisateur souhaite insérer l'enregistrement de clé 11 dans l'arbre B+ N°1, sans créer de nouvel arbre. Le chemin menant de la racine à la feuille où insérer l'enregistrement de clé 11 dans l'arbre B+ N°1 est : (n_0, f_2). Or la valeur du nœud générique f_2 associée à l'arbre B+ N°1 est $(8, 9, 10)$. La feuille f_2 est donc pleine dans l'arbre B+ N°1. Sa sœur gauche f_1 a pour valeur $(4, 5, 6)$ dans l'arbre B+ N°1 ; elle est également pleine. Par conséquent, il faut créer une nouvelle feuille générique f_4. Cette feuille est d'abord créée vide ; la valeur $\perp$ est associée à l'arbre B+ N°1, racine de la Hiérarchie d'Arbres. Puis, les clés 8 à 11 sont répartie entre f_2 et f_4 dans l'arbre B+ N°1 de sorte que la valeur de la feuille générique f_2 associée à l'arbre B+ N°1 soit $(8, 9, -)$ (avec comme feuille suivante f_4) et que la valeur de la feuille générique f_4 associée à l'arbre B+ N°1 soit $(10, 11, -)$. La valeur de f_2 est modifiée par l'insertion, par conséquent le partage implicite entre l'arbre B+ N°1 et ses descendants dans la Hiérarchie d'Arbres doit être supprimé. Pour cela, l'identificateur de l'arbre B+ N°2 est explicitement associé à la valeur $(8, 9, 10)$ de f_2. De même, les arbres B+ N°2 et 3 sont explicitement associés à la valeur $\perp$ de f_4, après modification de cette dernière dans l'arbre N°1. Comme le nœud n_0 est plein dans l'arbre B+ N°1, un nouveau nœud générique n_3 est créé. Il est d'abord créé vide ; la valeur $\perp$ est associée à la racine de la Hiérarchie d'Arbres, l'arbre B+ N°1. Puis, le nœud n_3 est mis à jour de sorte que ses fils dans l'arbre B+ N°1 soient f_2 et f_4 (les feuilles étant réparties entre les nœuds n_0 et n_3). L'insertion éclate les nœuds jusqu'à la racine. La racine générique est mise à jour. La figure 7.15 représente l'Arbre B+ Générique après l'insertion de l'enregistrement de clé 11 dans l'arbre B+ N°1.

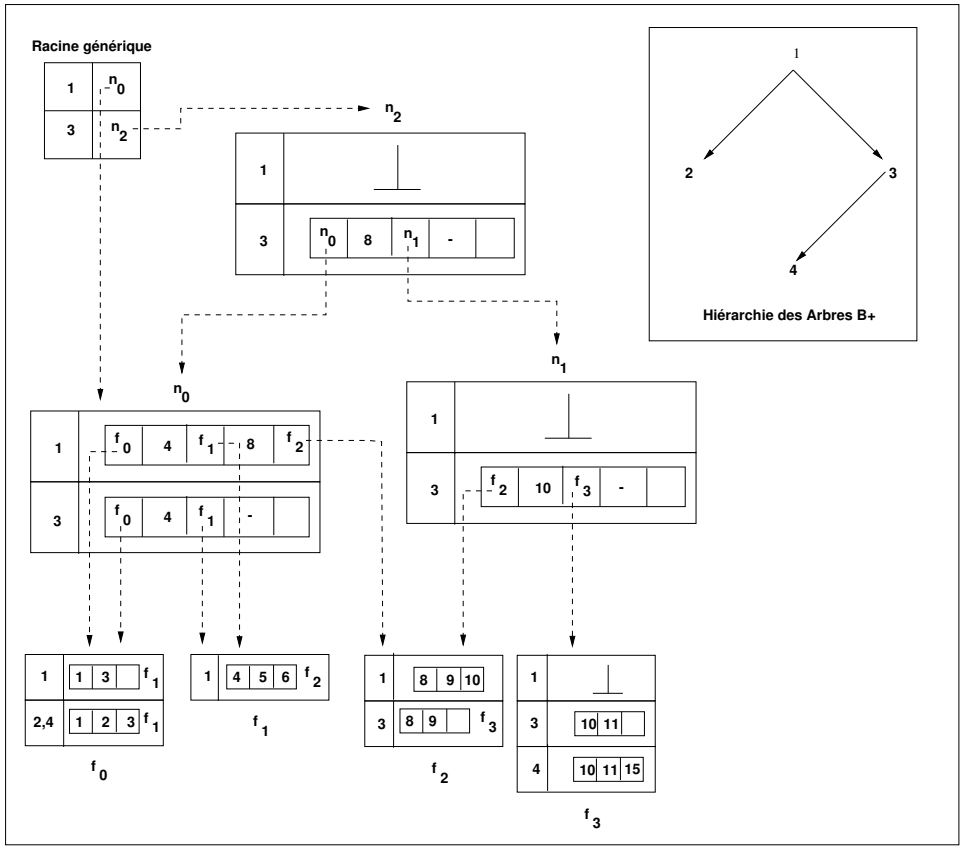

FIG. 7.14 – **L'Arbre B+ Générique après l'insertion des enregistrements de clés** 2 **et** 15 **dans les arbres B+ N°**4.

Exemple 7.15 L'utilisateur souhaite insérer la clé 12 dans l'arbre B+ N°2, sans créer de nouvel arbre B+. L'algorithme de recherche retourne le chemin (n_0, f_2). Or La feuille f_2 est pleine dans l'arbre B+ N°2, de même que sa feuille sœur f_1. Il faut donc éclater la feuille f_2. Or l'arbre B+ N°2 est fils de l'arbre B+ N°1 dans la Hiérarchie d'Arbres. Par conséquent, on regarde si f_2 a une feuille sœur droite dans l'arbre N°1. La feuille f_4 est sœur de f_2 dans l'arbre N°1 (voir valeur associée à l'arbre N°1 dans le nœud générique f_2) et la valeur de f_4 associée à l'arbre B+ N°2 est $\perp$, ce qui signifie que la feuille f_2 a été éclatée dans l'arbre N°1, père de l'arbre B+ N°2 dans la Hiérarchie d'Arbres. Par conséquent, l'éclatement de f_2 dans l'arbre B+ N°2 ajoute une nouvelle ligne dans le nœud générique f_4 de valeur $(10, 12, -)$. L'arbre B+ N°2 doit être associé à la valeur $(8, 9, -)$ du nœud générique f_2. Cette valeur existe et est associée à l'arbre B+ N°1. Par conséquent, la suppression de l'identificateur de l'arbre B+ N°2 dans le nœud générique f_2 permet d'associer implicitement l'arbre B+ N°2 à la valeur $(8, 9, -)$. Le nœud n_0 est plein dans l'arbre B+ N°2. Il doit donc être éclaté. Or le nœud n_0 ne référence pas la feuille f_4 dans la valeur associée à l'arbre B+ N°1, il a donc également été éclaté dans l'arbre B+ N°1. Par conséquent, le père de f_4 dans l'arbre B+ N°1 est recherché : il s'agit de n_3. La valeur de n_3 après éclatement du nœud n_0 dans l'arbre B+ N°2 est la même que celle de l'arbre B+ N°1, la valeur $< f_2, 10, f_4, -, >$ est donc implicitement partagée par les arbres B+ N°1 et 2. Il en est de même pour la valeur de

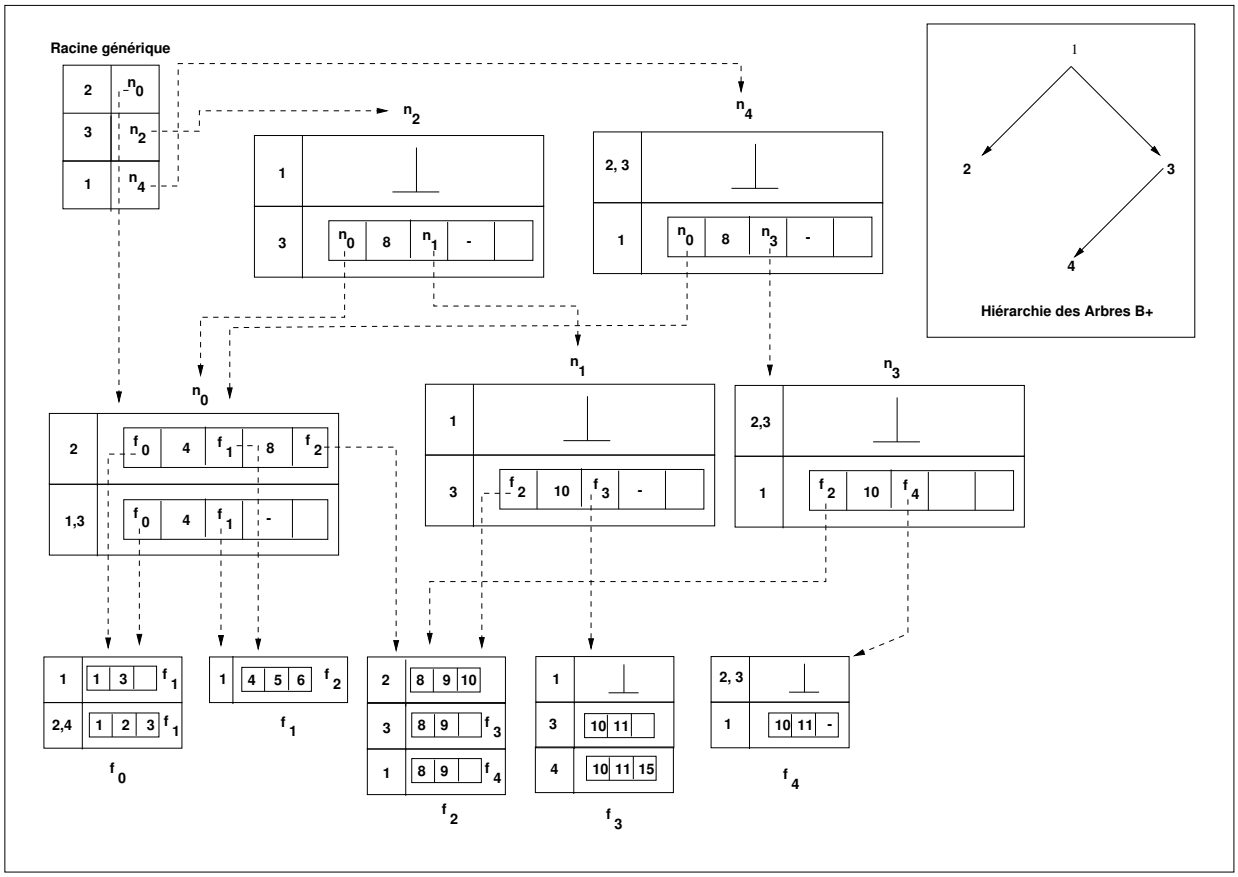

FIG. 7.15 – **L'Arbre B+ Générique après l'insertion de l'enregistrement de clé** 11 **dans l'arbre B+ N°1.**

la racine générique, la ligne associant le nœud n_0 à l'arbre B+ N°2 étant supprimée.

La suppression d'un ou plusieurs enregistrements dans un arbre B+ stocké dans un Arbre B+ générique n'est pas décrite dans ce chapitre. Pour réaliser une suppression, il suffit de remplacer l'éclatement de nœud, qui entraîne éventuellement la création de nœuds génériques, en fusion de nœuds qui peut entraîner la suppression de nœud générique.

7.3.2 Rechercher et insérer dans un Arbre R Générique

a) Pointé sur un point P dans un arbre R a_i

Comme pour l'Arbre B+ Générique, l'opération de pointé dans un Arbre R Générique diffère peu de l'opération de pointé dans un arbre R classique. Les seules différences sont :

1. La racine de l'arbre R a_i doit être recherchée : il faut pour cela rechercher à quelle valeur de la racine générique est associé l'arbre R a_i.

2. Chaque lecture d'un nœud n est précédée par la recherche, dans le nœud générique n, de la valeur de n dans l'arbre R a_i.

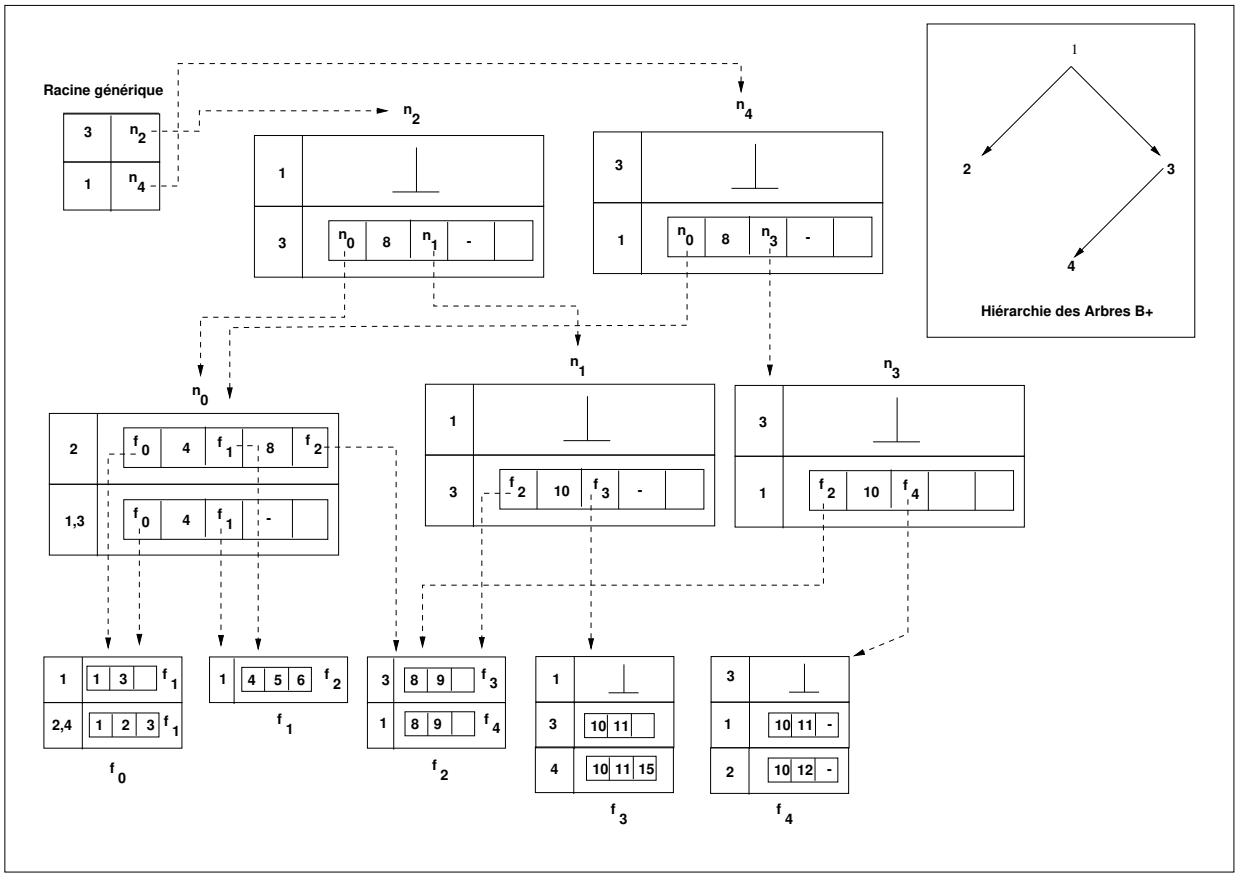

FIG. 7.16 – **L'Arbre B+ Générique après l'insertion de l'enregistrement de clé** 12 **dans l'arbre B+ N°2.**

Exemple 7.16 Le pointé sur le point P (voir figure 7.8 page 111) dans l'arbre R N°2, stocké dans l'Arbre R Générique de la figure 7.12 (voir page 118) est réalisé de la manière suivante. La racine générique est lue afin de connaître le nœud racine de l'arbre R N°2. Comme l'identificateur 2 n'apparaît pas, on déduit que le nœud racine est le même dans l'arbre R N°2 et dans l'arbre R N°1, ancêtre de l'arbre N°2 dans la Hiérarchie d'Arbres. Le nœud r est donc racine de l'arbre N°2. Les fils de r dans l'arbre N°2 sont a, b, c et d (voir nœud générique r de la figure 7.12). Le point P est inclus dans les rectangle englobants minimums de c et d. La valeur de ces nœuds associée à l'arbre R N°2 doit être recherchée dans le nœud générique correspondant. On déduit que le point P est inclus dans les rectangles englobants minimums des objets 12 et 13.

b) Insertion d'un objet o_k dans un arbre R a_i

Comme pour l'Arbre B+ Générique, il est possible de créer un nouvel Arbre R pour enregistrer des mises à jour ou de faire des mises à jour directement sur un arbre R.

Les différences entre l'algorithme d'insertion classique et l'algorithme d'insertion dans un arbre R stocké dans un Arbre R Générique sont les mêmes que celles qui existent entre l'algorithme

d'insertion dans un arbre B+ classique et dans un Arbre B+ Générique. Seuls des exemples sont donc traités dans cette sous-section.

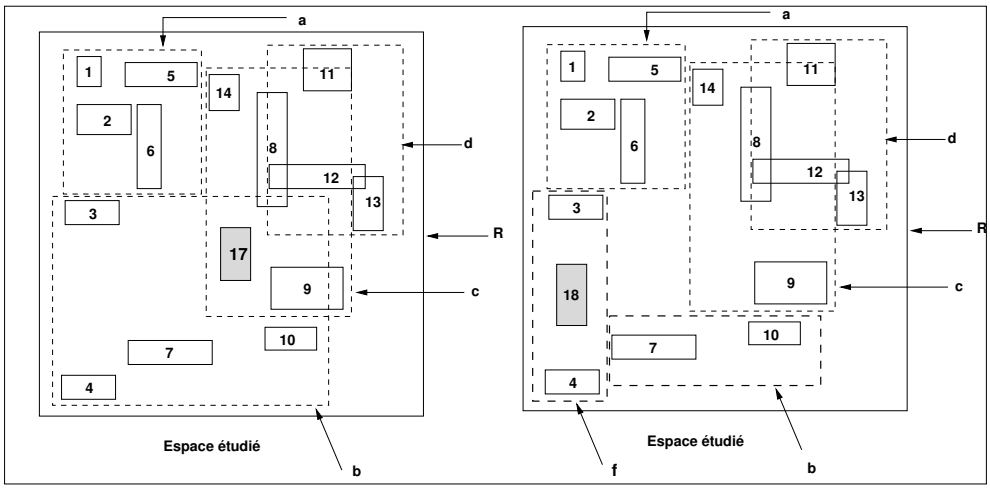

FIG. 7.17 – **L'insertion de deux objets** 17 **et** 18 **dans l'espace représenté par l'arbre R N°1.**

Exemple 7.17 L'utilisateur souhaite insérer un objet 17 dans l'espace représenté par l'arbre R N°1 (voir figure 7.17 à gauche). Il ne souhaite pas perdre l'arbre R N°1. Par conséquent il crée un nouvel arbre R, numéroté par 4, identique à l'Arbre R N°1, par insertion de l'identificateur 4 dans la Hiérarchie d'Arbres : l'arbre R N°1 est copié logiquement. Puis l'Arbre R N°2 est mis à jour en insérant l'objet 17. L'algorithme de recherche montre que l'objet 17 doit être inséré dans le nœud c sans modifier le rectangle englobant minimum de c. La valeur de c associée à l'arbre R N°4, par partage implicite avec l'arbre N°1, est $[8, 9, 14]$. Le nœud c n'est donc pas saturé. Une nouvelle ligne est ajoutée au nœud générique c. Cette ligne associe la valeur $[8, 9, 14, 17]$ à l'arbre R N°4. Comme l'arbre R N°4 est feuille de la Hiérarchie d'Arbres, aucune suppression de partage implicite n'est nécessaire. La figure 7.18 représente l'Arbre R Générique après insertion de l'objet 17 dans l'arbre R N°4.

Exemple 7.18 L'utilisateur souhaite insérer l'objet 18 directement dans l'arbre R N°1, qui est stocké dans l'Arbre R Générique de la figure 7.18. L'objet 18 doit être inséré dans le nœud b qui est saturé dans l'arbre R N°1. En effet, la valeur de b associée à l'arbre R N°1 est $[3, 4, 7, 10]$. Il faut donc diviser le nœud b en deux. Cette division de b donne naissance au nœud f et donc au nœud générique f. Le nœud générique f est tout d'abord créé vide, c'est-à-dire avec la valeur $\perp$ associée à la racine de la Hiérarchie d'Arbres, l'arbre N°1. Puis, la valeur du nœud b est mise à jour dans l'arbre R N°1 ; une nouvelle ligne de valeur $[3, 4, 18]$ est insérée dans le nœud générique b. Cette valeur est associée à l'arbre R N°1. Le partage implicite entre l'arbre R N°1 et ses descendants dans la Hiérarchie d'Arbres, c'est-à-dire les arbres R N°2 et 4, est supprimé : les arbres R N°2 et 4 sont explicitement associés à la valeur $[3, 4, 7, 10]$ du nœud générique b. De la même manière, une nouvelle ligne est ajoutée au nœud générique f associant la valeur $[7, 10]$ à l'arbre R N°1, le partage implicite avec ses descendants ayant été supprimé. Comme le nœud r est saturé dans

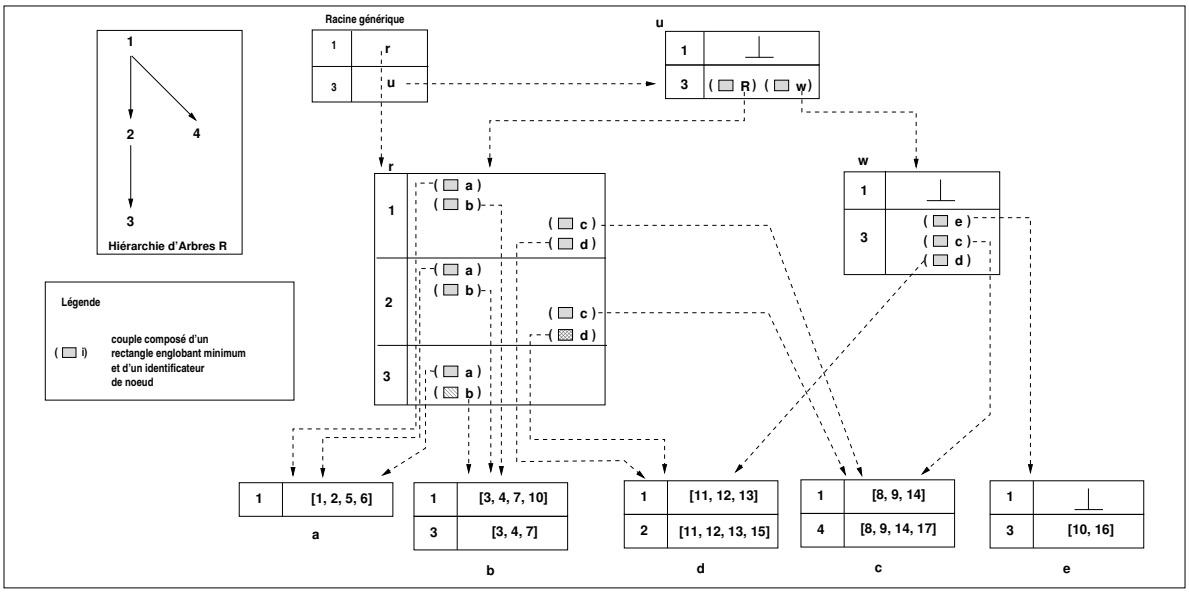

FIG. 7.18 – **L'Arbre R Générique après l'insertion de l'objet** 17 **dans l'arbre R N°**4.

l'arbre R N°1, il est divisé en deux, créant deux nouveaux nœuds y et x. La racine générique est mise à jour de sorte que le nœud x soit racine dans l'arbre R N°1, et que le nœud r demeure racine des arbres R N°2 et 4.

7.4 Applications

Cette section présente le cadre dans lequel s'appliquent les structures d'Arbre B+ et d'Arbre R Génériques. La sous-section 7.4.1 montre comment ces structures s'intègrent à l'Approche des Versions de Bases de Données, présentée dans le chapitre 2 de la partie A. La sous-section 7.4.2 présente l'utilisation de ces structures pour gérer et interroger des versions de collections. La sous-section 7.4.3 décrit les perspectives envisagées pour appliquer ces structures à la gestion des versions de BLOB ou des versions de documents textuels ou hyper-textuels.

7.4.1 Approche de versions unifiée

Les structures d'Arbre B+ et d'Arbre R Génériques complètent l'édifice construit par les différents travaux sur l'approche des Versions de Bases de Données [CJ90, BMJ94, CJV94, Gan94, GJ96, DGJM96a, DGJM96b, Abd97, AJ97a, GRW97, Mon97, BJC98, Zam98, ZJ99]. L'approche des Versions de Bases de Données est une approche unifiée, dans le sens où toutes les entités, c'est-à-dire les objets [CJ90], les n-uplets [Abd97], les contraintes [DGJM96a, DGJM96b, Mon97] et le schéma [BJC98], sont gérées avec les mêmes mécanismes de versionnement. Les structures d'Arbre B+ et d'Arbre R Générique ajoutent, à cette approche, la gestion d'index permettant notamment de gérer plus facilement des versions de collection d'objets.

Dans l'approche des Versions de Bases de Données de [CJ90, CJ98], une collection d'objets possède

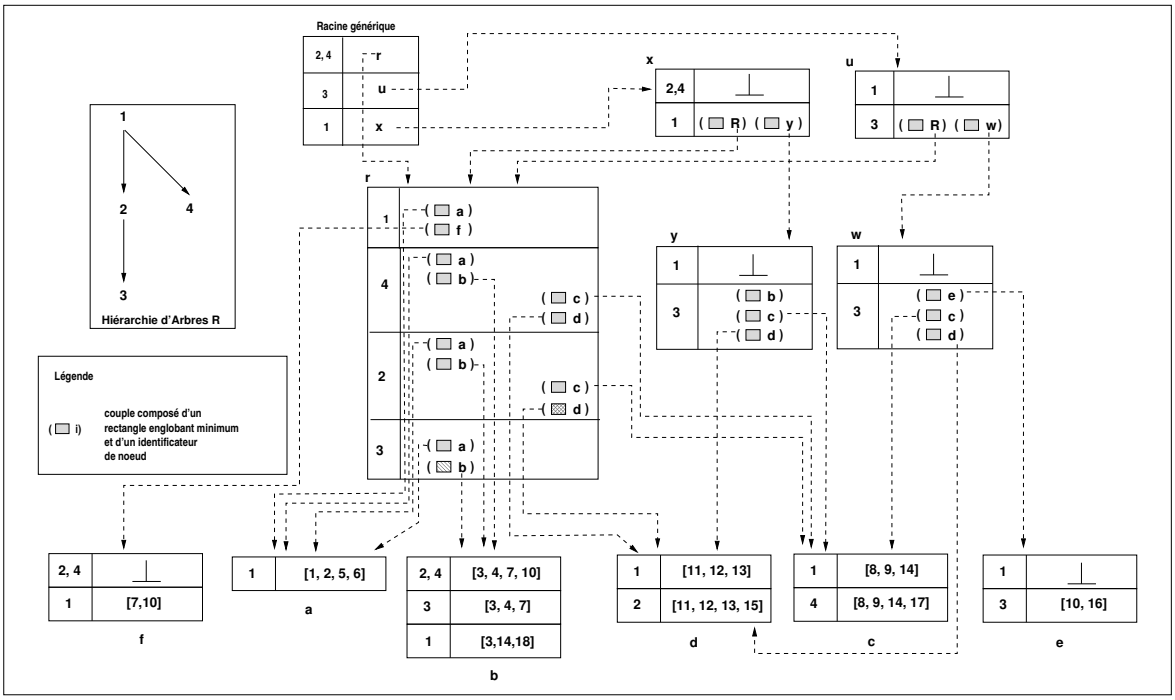

FIG. 7.19 – **L'Arbre R Générique après l'insertion de l'objet** 18 **dans l'arbre R N°**1.

une version logique dans chaque VBD de la base. Une version logique de collection regroupe des versions logiques d'objet dans une VBD. Le nombre d'éléments, c'est-à-dire de versions logiques d'objet, ainsi que les éléments mêmes, peuvent différer d'une version de la collection à une autre.

Exemple 7.19 La figure 7.20 représente une collection de personnes (objets de la classe *Personne*) dans une base de données multiversion, composée de trois Versions de Bases de Données, v_1, v_2 et v_3. Lorsque le nombre d'éléments dans la collection est important, il est souhaitable de disposer d'une structure de données particulière pour accéder aux versions logiques d'objets contenues dans les versions de la collection *LesPersonnes*. Une telle structure permet, par exemple, de répondre aux requêtes suivantes : *"Quelles différences y a t - il, dans la collection LesPersonnes, entre les VBD v_1 et v_2 ?"* ou *"Paul a t - il toujours appartenu à cette collection?"*, ou encore *"Quels éléments de la collection apparaissent dans les VBD v_1 et v_3, mais pas dans v_2 ?"*, etc.

Le type d'index multiversion, décrit dans l'exemple précédent, répond principalement à quatre besoins : [1] Retrouver facilement les éléments d'une (ou plusieurs) version(s) de collection ou de tester l'appartenance d'un élément à une (ou plusieurs) version(s) ; [2] Faciliter la comparaison entre versions de collection ; [3] Permettre de faire des mises à jour sur toutes versions de collection ; [4] Limiter la redondance entre versions de collection ; [5] Etre indépendant de la sémantique liée au versionnement pour s'adapter à tout type d'applications.

Les sous-sections qui suivent montrent que les Arbres B+ et R Génériques permettent de répondre à ces besoins.

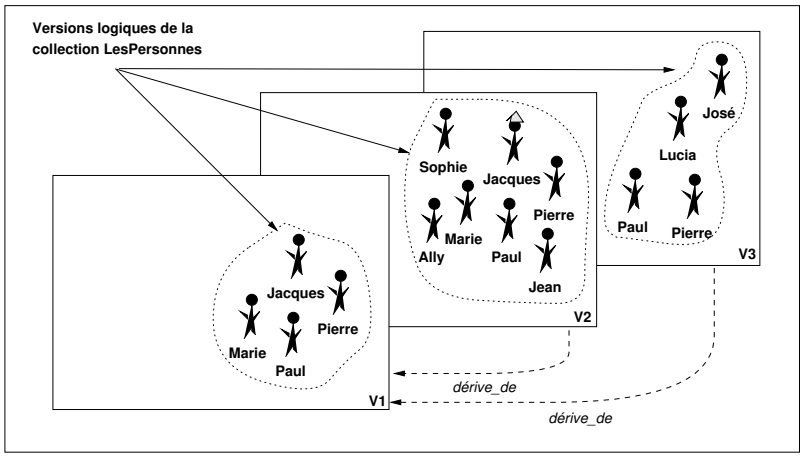

FIG. 7.20 – **Un exemple de versions de collection dans une base de données multiversion**.

7.4.2 Gestion des versions de collection d'objets

Dans les sous-sections qui suivent, on suppose qu'on utilise un Arbre B+ Générique pour gérer des versions d'une collection d'objets $\mathcal{C}$. On suppose également que chaque version v_i de la collection $\mathcal{C}$ est indexée par un arbre B+ a_i. La collection $\mathcal{C}$ est indexée par un Arbre B+ Générique regroupant tous les arbres B+ a_i. Dans cette sous-section, le cas de l'Arbre R Générique n'est pas traité, puisque sa gestion est la même que celle de l'Arbre B+ Générique.

a) Test d'appartenance d'un objet à une ou plusieurs versions de collection

Pour tester l'appartenance d'un objet à une version de collection, il suffit :

- D'associer à chaque version de la collection un arbre B+ dont les clés d'index sont les identificateurs des objets éléments de la version de collection.
- D'indexer la collection par un Arbre B+ Générique regroupant tous les arbres B+ associés aux versions de la collection.

Exemple 7.20 Supposons que l'Arbre B+ Générique de la figure 7.11 (voir page 116) indexe une collection qui possèdent trois versions respectivement indexées par les arbres B+ N°1, 2 et 3. Les clés d'index correspondent aux identificateurs des éléments de la collection.

Le test d'appartenance d'un objet o de clé k à une version v_i de collection indexée par un arbre B+ a_i revient à appliquer l'algorithme de recherche décrit dans la section 7.3.1.

Pour tester l'appartenance d'un objet o de clé k à plusieurs versions de collection v_1 ... v_n, il suffit d'appliquer l'algorithme de recherche dans la version de collection v_1. L'algorithme retourne le nœud feuille f de l'arbre a_1, indexant v_1, qui doit contenir l'objet de clé k. Puis, il suffit de lire quelques nœuds génériques dans l'Arbre B+ Génériques, pour tester l'appartenance de l'objet de clé k aux autres versions.

De manière plus précise, le test d'appartenance d'un objet o de clé k à plusieurs versions de collection v_1 ... v_n, est réalisé de la manière suivante :

1. L'objet o de clé k est recherché dans la version v_1 de la collection : le chemin menant de la racine à la feuille f, où doit se trouver l'objet dans l'arbre B+ a_1, est mémorisé. La clé k est recherchée dans la valeur de f associée à l'arbre a_1.
2. Puis, pour chaque arbre a_i, on regarde, dans le nœud père n de f, s'il faut chercher la clé k dans la feuille f :
   (a) Si f mène à la clé k dans l'arbre a_i, alors la clé k est recherchée dans la valeur de f associée à l'arbre a_i.
   (b) Si une autre feuille f' mène à la clé k dans l'arbre a_i, alors la clé k est recherchée dans la valeur de f' associée à l'arbre a_i.
   (c) Si aucune feuille du nœud n ne permet d'accéder à la clé k, alors l'agorithme reprend avec le nœud père de n, afin de trouver le chemin correct menant à la clé k dans l'arbre a_i. L'algorithme peut éventuellement remontrer dans l'arbre jusqu'à la racine. Cependant, si les arbres B+ ont peu de différences les uns par rapport aux autres, l'algorithme ne va lire que quelques nœuds génériques.

En fait, chaque fois que l'on accède à une feuille f partagée par plusieurs arbres B+ a_1, ..., a_p, l'appartenance ou non de la clé k à f permet de conclure l'appartenance ou non de la clé k aux p collections v_1, ..., v_p correspondantes aux différents arbres. Il est inutile, si le partage entre les arbres B+ est conséquent, de parcourir tous les arbres B+ a_i pour cette réaliser cette opération.

Exemple 7.21 Supposons que l'utilisateur souhaite tester l'appartenance de l'objet de clé 10 dans les trois versions de la collection indexée par l'Arbre B+ Générique de la figure 7.11 (voir page 116). L'algorithme de recherche de l'objet de clé 10 dans l'arbre B+ N°1 indexant la version v_1 retourne la feuille f_2. L'objet de clé 10 appartient à la valeur $(8, 9, 10)$ de f_2 associée à l'arbre B+ N°1. Cette valeur est implicitement partagée par l'arbre B+ N°2. Par conséquent, on déduit que l'objet de clé 10 appartient également à la version v_2 de la collection $\mathcal{C}$. En revanche, la clé 10 n'apparaît pas dans la valeur de f_2 associée à l'arbre B+ N°3. La feuille suivante de f_2 dans l'arbre B+ N°3 est f_3. La valeur de f_3 dans l'arbre B+ N°3 est recherché. Finalement, on déduit que l'objet 10 appartient aux trois versions de la collection $\mathcal{C}$.

b) Comparaison de versions de collection

L'Arbre B+ Générique facilite le calcul de la différence de deux versions de collection, par le mécanisme de partage implicite. Soient deux versions v_i et v_j, indexées par les arbres B+ a_i et a_j stockés dans un Arbre B+ Générique. Pour comparer les deux versions de la collection, il faut procéder ainsi :

1. Rechercher la première feuille f de l'un des deux arbres, par exemple a_i. Soit $\mathcal{D}$ l'ensemble des objets n'appartenant pas aux deux versions de la collection. Au début de l'algorithme $\mathcal{D}$ est vide.

2. Puis, rechercher la valeur de f associée à a_i :

   (a) Si la valeur de f est implicitement ou explicitement partagées par les deux arbres a_i et a_j, il faut reprendre l'algorithme à l'étape 2 avec la feuille située à droite de f.

   (b) Si f a une valeur v dans l'arbre a_i et une valeur v' dans l'arbre a_j, avec $v \neq v'$, il faut mettre à jour l'ensemble $\mathcal{D}$ tel que : $\mathcal{D} = [\mathcal{D} \cup (v \cap v')] \setminus [\mathcal{D} \cap (v \cap v')]$. Puis :

      i. Si la feuille située à droite de f est la même dans les deux arbres B+ a_i et a_j, alors il faut reprendre l'algorithme à l'étape 2 avec la feuille sœur de f.

      ii. Si la feuille située à droite de f est f_i dans l'arbre a_i et f_j dans l'arbre a_j, $f_i \neq f_j$, alors il faut comparer les valeurs respectives des feuilles f_i et f_j dans les arbres B+ a_i et a_j et reprendre l'algorithme à l'étape 2 avec chacune des feuilles f_i et f_j.

Exemple 7.22 La différence des versions v_2 et v_3 de la collection indexée par l'Arbre B+ Générique de la figure 7.11 (voir page 116) est calculée de la manière suivante :

1. La première feuille de l'arbre B+ N°2 est f_0.

2. Une valeur de f_0 est associée à chaque arbre B+ N°2 et 3 (l'arbre N°3 partage implicitement la valeur associée à l'arbre B+ N°1). En comparant ces valeurs, on déduit que $\mathcal{D} = \{2\}$.

3. La feuille f_1 est située à droite de f_0 dans les deux arbres B+. La valeur de f_1 dans chaque arbre est comparée. Cette valeur est explicitement partagée par les deux arbres. La valeur de $\mathcal{D}$ est inchangée.

4. La feuille f_2 est située à droite de f_1 dans les deux arbres B+. La valeur de f_1 dans chaque arbre est comparée. La valeur de f_2 associée à l'arbre B+ N°2 est $(8, 9, 10)$ et celle associée à l'arbre B+ N°3 est $(8, 9, -)$. Par conséquent, on déduit que $\mathcal{D} = \{2, 10\}$.

5. La feuille suivante de f_2 est f_3 dans l'arbre B+ N°3. La feuille f_3 n'existe pas dans l'arbre B+ N°2, la valeur associée à l'arbre étant N°2 $\perp$. On déduit que $\mathcal{D} = (\{2, 10\} \cup \{10, 11\}) \setminus (\{2, 10\} \cap \{10, 11\}) = \{2, 11\}$

La différence entre les versions v_2 et v_3 de la collection correspond donc aux éléments de clé 2 et 11.

La place occupée par les nœuds interne d'un arbre B+ occupe en général 1% de la place occupée par les nœuds feuilles [Lom98]. Les mises à jour (ajout, insertion ou suppression d'enregistrements dans un arbre B+) touchent, par conséquent, plus les nœuds feuilles que les nœuds internes. Dans ce cas, l'Arbre B+ Générique facilite la comparaison des versions de collection. En effet, les nœuds génériques permettent de naviguer à travers les différentes valeurs d'un même nœud.

c) Mises à jour de versions de collection

La section 7.3.1 a présenté les algorithmes de recherche et d'insertion dans un arbre B+ stocké dans un Arbre B+ Générique. Les mises à jour peuvent se faire sur n'importe quel arbre B+ de l'Arbre Générique et donc sur n'importe quelle version de la collection. L'utilisateur peut choisir à tout moment de créer une nouvelle version (ou non) et donc un nouvel arbre B+, suite aux mises à jour.

d) Indépendance de la sémantique liée au versionnement

Les Arbres Génériques se basent sur l'approche des Versions de Bases de Données. Il peuvent néanmoins être utilisés dans n'importe quelle approche de versionnement. Les liste d'identificateurs associés à chaque valeur de nœud peuvent être soit une estampille temporelle, soit un identificateur d'arbre B+, un identificateur de version ou un identificateur de VBD ou de contexte.

7.4.3 Perspectives

Dans les bases de données, les objets de grande taille sont généralement stockés sous forme de BLOB [KB96, BN98]. Les documents XML, par exemple, sont dans la pratique décomposés en fragments correspondants aux entités réutilisables les plus fines ; ces entités sont stockées sous forme de BLOB dans la base [Mic98]. Ces BLOB peuvent être gérés par des arbres B+ [CDRS89, Bil92b]. Ils peuvent, comme les documents textuels ou hyper-textuels, posséder des versions [CDRS89, BLPS96, KB96, OHE96, BN98, Mic98]. Par conséquent, l'Arbre B+ Générique peut être envisagé pour gérer ce type de versions d'objet de grande taille peu ou pas structurés, en offrant des mécanismes de partage plus souples que le stockage différentiel proposé dans la plupart des approches [CDRS89, VD95, WL97, BN98]. Cependant, la mise à jour d'un tel objet peut complètement modifier la structure de l'arbre. Il faut donc réfléchir à l'adaptation des mécanismes proposés pour répondre à ce problème et permettre un partage efficace.

Il serait également intéressant d'utiliser les Arbres Génériques dans le domaine des entrepôts de données (*datawarehouse*). En effet, les entrepôts de données collectent souvent des informations redondantes. Par exemple, on suppose qu'un entrepôt de données stocke chaque jour le prix des produits d'un magasin. Peu de produits changent quotidiennement de prix. Cependant, chaque jour, une nouvelle version des prix est créée. De manière générale dans les entrepôt, les versions sont stockées sans partage. Or, si seulement 5% des prix changent quotidiennement, un espace mémoire énorme va être perdu au bout de quelques mois ou de quelques années. Par conséquent, il est envisageable d'indexer les relations stockées dans l'entrepôt par un Arbre Générique, afin de ne pas répéter des n-uplets communs à plusieurs versions.

D'autres perspectives envisageables pour les Arbres Génériques sont présentées en conclusion de la thèse.

Conclusion

Les mécanismes proposés pour l'Arbre Quaternaire Générique ont été généralisés, dans ce chapitre, aux arbres B+ et aux arbres R. L'arbre quaternaire est un arbre non équilibré dont le nombre de nœuds fils par nœud d'index est de zéro ou quatre. Le nombre de niveaux d'un arbre quaternaire croît par les feuilles (un nœud feuille peut être découpé en quatre sous-nœuds) . Les arbres B+ et R en revanche, croissent par la racine et n'ont pas un nombre fixe et prédéfini de nœuds fils par nœud d'index. Par conséquent, les mécanismes proposés permettent de gérer des familles d'arbres, quel que soit le type des arbres. Ces mécanismes peuvent s'appliquer aux arbres dont la structure est similaire à celle des arbres quaternaires, comme les arbres K-D, ou aux arbres dérivées des arbres B+ et R.

Cette généralisation aux structures arborescentes permet en outre de gérer non seulement des objets de grande taille faiblement structurés comme les images, mais aussi des objets de grande taille fortement structurés comme les collections d'objets. Ce chapitre a présenté la gestion des versions de collection d'objets par un Arbre B+ Générique. Cette structure permet de tester l'appartenance d'un objet à une ou plusieurs versions de collection, de comparer des versions de collection et de mettre à jour n'importe quelle version de collection, en laissant à l'utilisateur le choix de dériver ou non une nouvelle version suite aux mises à jour. Elle complète le modèle unifié de l'approche des Versions de Bases de Données et peut également s'adapter à tout modèle de versions ; la structure est indépendante de la sémantique du versionnement. Le chapitre qui suit compare les Arbres Génériques proposés avec les approches de la littérature, dans le cadre de la gestion de versions de collection.

Chapitre 8

Comparaison des Arbres Génériques avec les autres travaux

Résumé

Ce chapitre compare les Arbres B+ et R Génériques avec les modèles d'indexation de versions de collection, une collection pouvant correspondre à une collections de n-uplets, une collection d'objets ou la base toute entière vue comme une collection de données. Les travaux de recherche peuvent être classés en trois catégories : les index à une dimension, les index à deux dimensions et, parmi les index à deux dimensions, les structures dites persistantes. Un index à une dimension indexe les versions de collection en fonction d'une clé d'index indépendante ou non de la sémantique du versionnement. Cette clé peut être un identificateur d'entité, permettant de savoir de quelles versions de la collection cette entité est élément, ou un identificateur de versions, les entités étant dans ce cas regroupées par versions de collection. Un index à deux dimensions indexe les versions de collection par un couple (attribut, identificateur de version), l'attribut étant indépendant de la sémantique de versionnement. Dans cette catégorie d'index, il existe des structures dites persistantes, c'est-à-dire des structures d'index qui possèdent des versions.

Introduction

Les modèles de versions proposés [KBG89, KBC+89, KS92, Loo92, Haa94] n'emploient pas, à notre connaissance, le terme de versions de collection d'objets, exceptés [LF91] et [Rei97]. Le modèle du système *CAMERA* de [LF91] gère les versions de collection de différentes manières. Lorsque le versionnement est linéaire, dans le cas temporel par exemple, un couple (*naissance*, *mort*) est associé à chaque élément e de la collection. Ce couple signifie que l'élément e est élément de la collection pendant l'intervalle de temps [*naissance*,*mort*]. Lorsque le versionnement n'est pas linéaire (une version a plusieurs successeurs), les auteurs de [LF91] proposent deux manières de gérer les versions de collection en fonction du nombre d'éléments de la collection. Lorsque la cardinalité de la collection est petite, un mécanisme de stockage différentiel est utilisé. En revanche, dès que la cardinalité est grande, chaque version de la collection est représentée par un arbre AVL, c'est-à-dire un arbre binaire équilibré [Knu73]. Des mécanismes de chevauchement (voir chapitre 1, partie A) sont appliqués aux arbres AVL pour optimiser le stockage.

Les index facilitent l'interrogation des collections, notamment lorsque la cardinalité de la collection est grande [DLR91, YM98a]. Trois grands groupes de travaux sur l'indexation des versions de collection peuvent être distingués [DSST89, Rei97] : les index à une dimension, les index à deux dimensions et les structures persistantes. Un index à une dimension permet d'accéder aux éléments des versions de collection via une clé d'index, qui peut être ou non indépendante de la sémantique du versionnement. Un index à deux dimensions permet, quant à lui, d'accéder aux éléments en fonction de l'identificateur de la version de collection et d'une clé d'index non liée à la sémantique du versionnement (ex. les identificateurs des éléments). Les structures dites persistantes [DSST89] peuvent être considérées comme des index à deux dimensions particuliers qui associent une version de l'index à chaque version de la collection.

Ce chapitre présente différents modèles d'indexation de versions de collection. L'objectif est de situer les structures d'Arbres B+ et R Génériques, présentées dans le chapitre 7, dans leur contexte scientifique. Une comparaison des Arbres Génériques avec les travaux proches est présentée à la fin du chapitre.

Les approches proposées [CDRS89, MK90, LF91, LM91, EWK93, Kol93, LS93, BGO+96, Rei97, VV97, TML99] se basent sur différents modèles de versions : versionnement linéaire où chaque version a exactement un successeur excepté la dernière, ou versionnement non linéaire où une version peut avoir un ou plusieurs successeurs. Les apports et les limites de ces propositions sont étroitement liés aux choix adoptés pour la représentation des versions. Dans la suite de ce chapitre, lorsque cela n'est pas précisé, une collection correspond indifféremment à une collection de $n-uplets$, une collection d'objets ou encore la base de données vue comme une collection d'entités.

Ce chapitre est organisé comme suit. La section 8.1 présente les mécanismes des index dits à une dimension. La section 8.2 décrits les structures d'index à deux dimensions. La section 8.3 présente plusieurs types de structures persistantes. Finalement, la section 8.4 compare les autres approches aux structures d'Arbres B+ et R Génériques. Les tableaux 8.1 (voir page 150), 8.2 (voir page 151), 8.3 (voir page 154) et 8.4 (voir page 155) résument les différentes approches de la recherche. Le tableau 8.5 (voir page 156) résume les apports des Arbres B+ et R Génériques.

8.1 Index à une dimension

Un index à une dimension permet d'accéder aux éléments des versions de collection en fonction d'une clé d'index. Cette clé d'index peut être indépendante de la sémantique du versionnement (voir en 8.1.1). Dans ce cas, l'index permet de savoir quelles sont les versions de collection pour lesquelles une entité a été élément. En revanche, lorsque la clé d'index est liée à la sémantique du versionnement (voir en 8.1.2), l'index regroupe les éléments par versions de collection.

8.1.1 Gestion d'un intervalle d'appartenance aux versions de collection

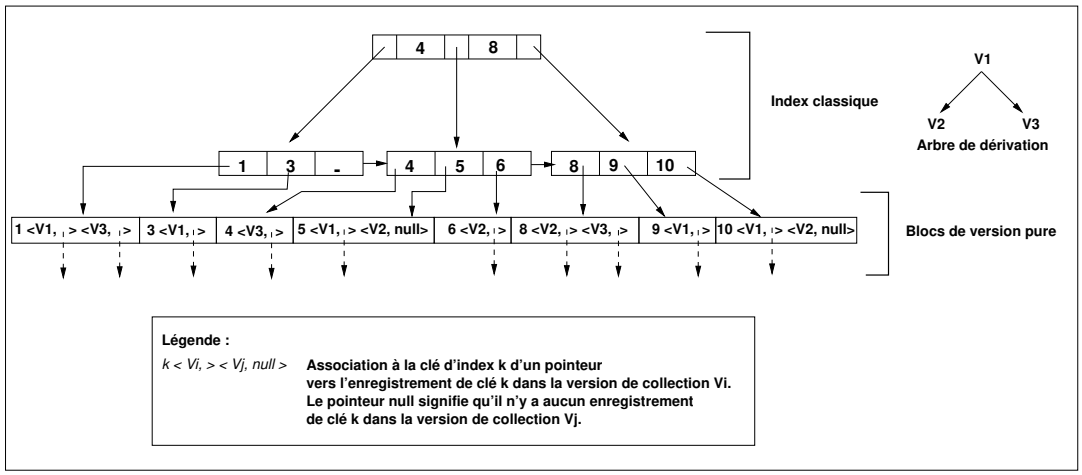

FIG. 8.1 – **Un exemple d'index à une dimension.**

Lorsque la clé d'index est indépendante de la sémantique du versionnement, à chaque élément de la collection est associé un intervalle d'appartenance à une ou plusieurs versions de la collection. Lorsque la sémantique du versionnement est le temps, l'intervalle d'appartenance est un intervalle de temps.

Dans l'approche de [LM91] (*The Pure Version Block Method*), l'index à une dimension est construit comme un index classique (ex. un arbre B+). Cependant, les feuilles de l'index pointent vers des pages, nommées *blocs de version pure* (ou *Pure Version Block*), qui contiennent des enregistrements de la forme *(k, <v_i, pointeur>$^+$)* ; le signe + signifie un ou plusieurs. A chaque valeur de clé k est associée un ou plusieurs couples *<v_i, pointeur>*, où *pointeur* pointe vers l'élément de clé k dans la version de collection v_i et, dans les versions dérivées de v_i si l'identificateur de ces versions n'apparaît pas dans le bloc de version pure. La valeur de *pointeur* peut être *null*, ce qui signifie que l'enregistrement de clé k a été supprimé de la version v_i.

Exemple 8.1 La figure 8.1 présente un exemple adapté de [LM91] de trois versions de collection identifiées de v_1 à v_3. La version v_2 a été créée à partir de la version v_1, par ajout des éléments de clé 6 et 8 et par suppression des éléments de clé 5 et 10. La version v_3 de la collection a été créée à partir de la version v_1, par ajout des éléments de clé 4 et 8 et par modification de la valeur de l'élément de clé d'index 1. L'élément de clé 1 a la même valeur dans les versions v_1 et v_2 de la

collection. Les éléments de clé 5 et 10 ont même valeur dans les versions v_1 et v_3 de la collection. Les éléments de clé 3 et 9 ont même valeur dans les trois versions de la collection.

L'avantage de ce type index est qu'il permet de savoir dans quelle version(s) de collection(s) une entité apparaît ou non et quelle est sa valeur. Cet index s'applique donc aux versions de collection dont les éléments peuvent également être versionnés. Cependant, il est difficile de faire des recherches par version de collection. En effet, pour retrouver tous les éléments d'une version de collection, il faut parcourir tout l'index et vérifier, pour chaque entité de la base, si elle appartient ou non à la version de collection recherchée.

8.1.2 Regroupement des éléments par versions de collection

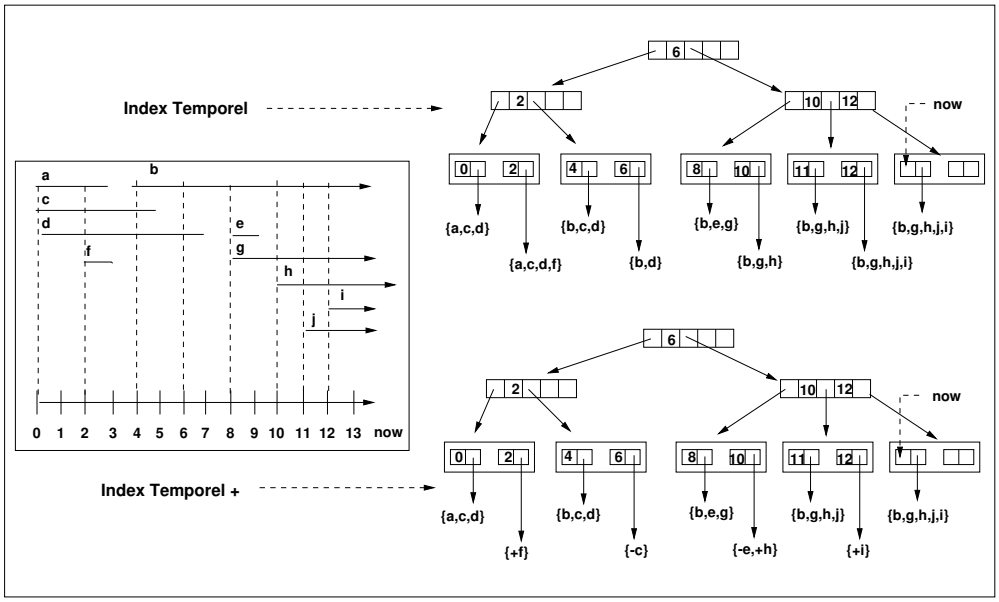

FIG. 8.2 – **Un exemple d'Index Temporel de [EWK93] et de sa variante.**

A l'inverse de l'approche présentée dans la sous-section précédente, l'approche de [EWK93] propose de regrouper les entités par versions de collection en gérant un index dont les clés dépendent de la sémantique de versionnement, qui dans cette approche est le temps. Cet index, nommé Index Temporel (*Time Index*), permet de retrouver les différents états d'une base de données en ajout seul (*append only database*), en fonction des intervalles de validité des entités de la base.

Le temps de validité est découpé en intervalles contigus, et chaque clé d'index est une date de validité correspondant au début d'un intervalle de validité ou à la fin d'un intervalle $+1$. Chaque feuille de l'index est composée de paires (k_i, p_i) telles que p_i pointe vers l'emplacement des entités valides pendant l'intervalle de temps $[k_i, k_{i+1}]$.

Exemple 8.2 La figure 8.2 présente un exemple d'Index Temporel (en haut de la figure). Les entités gérées dans la base et leur intervalle de validité respectif sont représentés sur le graphique

situé à gauche de la figure. Les clés de l'index sont 0, 2, 4, 6, 8, 10, 11, 12 et *now*, *now* correspondant à la date courante. La date 2 est clé d'index car l'intervalle de validité de l'entité f commence à la date 2. La date 6 est également clé d'index car l'intervalle de validité de l'entité c se termine à la date 5.
Les entités a, c et d sont valides pendant l'intervalle [0,2[. Elles sont référencées par le pointeur associé à la date 0 dans la première feuille de l'index. Pendant l'intervalle [2,4[, les entités a, b, c et f sont valides. Ces entités sont référencées par le pointeur associé à la date 2 dans la première feuille de l'index, etc.

Dans ce type d'approche, plus il y a d'intervalles de temps gérés dans la base, plus la taille de l'index est grande [ST94, Nas96]. Lorsque le nombre de versions d'entité par intervalle est grand, une deuxième structure d'accès semble nécessaire pour permettre d'accéder plus rapidement aux entité en fonction de leur identificateur. Il existe en outre une forte redondance au niveau des feuilles de l'index. En effet, les identificateurs d'entité sont recopiés de feuille en feuille, tant que les entités appartiennent à un intervalle de temps ou à une version de collection. Par exemple, sur la figure 8.2, tous les identificateurs d'entités valides pendant l'intervalle [2,*now*[sont recopiés dans toutes les feuilles dont la clé d'index est supérieure à 2. Pour répondre à ce problème, les auteurs de [EWK93] ont proposé une variante de l'Index Temporel, nommé Index Temporel +, qui ne stocke que les changements incrémentaux entre ensembles de versions d'entité accessibles via la même feuille d'index. Cependant, l'Index Temporel + impose une reconstruction, qui peut être longue, lorsque l'on veut savoir quelles sont les données valides pendant un intervalle ou à un moment.

Exemple 8.3 Cette proposition est illustrée en bas de la figure 8.2. Les entités identifiées par a, c, et d sont valides pendant les intervalles de temps [0,2[et [2,4[, alors que f n'est valide que pendant [2,4[. Dans l'Index Temporel (voir en haut de la figure), les identificateurs a, c, et d sont recopiés pour les deux intervalles. En revanche, lorsque les changements sont incrémentaux (voir Index Temporel + en bas de la figure), la clé 2 donne accès à l'ensemble $\{+f\}$, signifiant que f est valide pendant cet intervalle, mais que les éléments de l'intervalle précédent ([0,2[) le sont aussi. Le + signifie ici un ajout par rapport aux éléments de l'intervalle précédent. Un − signifie qu'il y a un élément de moins par rapport à l'intervalle précédent. Par exemple, pendant l'intervalle [4,6[, les entités b, c et d sont valides. L'ensemble $\{-c\}$ accessible par la clé 6 signifie que pendant l'intervalle [6,10[, seuls b et d restent valides par rapport à l'intervalle précédent.

Les caractéristiques de l'index proposé par [EWK93] sont résumées dans les tableaux 8.1 et 8.2 (pages 150 et 151). Ce type d'index permet de rechercher les valeurs des versions de collection, mais ne permet pas de rechercher les éléments en fonction d'une valeur d'attribut non liée à la sémantique de versionnement. Pour permettre ce genre de requête, les auteurs de [EWK93] proposent de construire un index classique sur une valeur d'attribut non temporel et d'associer à chaque feuille de cet index un Index Temporel. L'ensemble d'Index Temporels ainsi obtenu peut être assimilé à un index à deux dimensions. Les caractéristiques des index à deux dimensions sont présentées dans la section suivante.

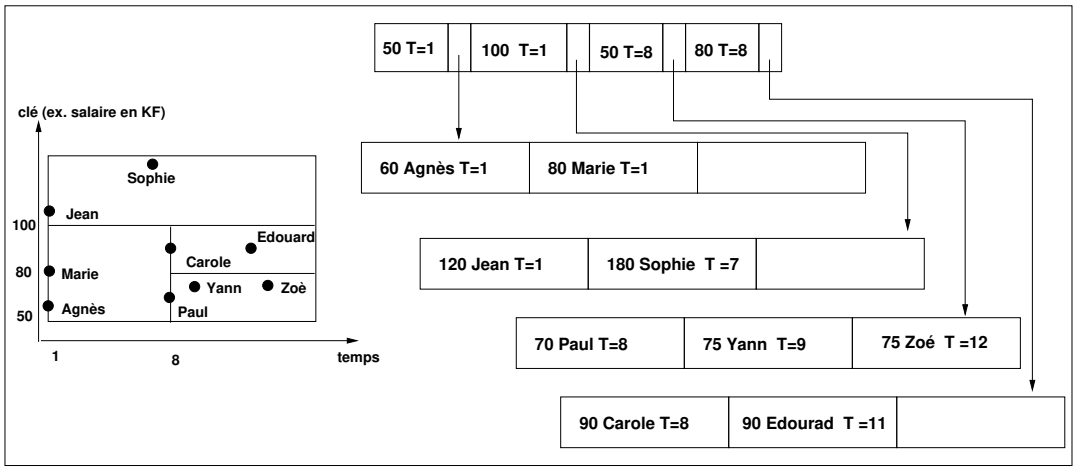

FIG. 8.3 – **Un exemple d'arbre TSB.**

8.2 Index à deux dimensions

Un index à deux dimensions permet d'accéder aux éléments de la collection en fonction d'une clé d'index indépendante à la sémantique du versionnement et d'un identificateur de version. Ces index peuvent être classés en deux catégories : ceux qui utilisent une clé formée d'un couple $(identificateur_de_version, attribut_non_temporel)$ et ceux qui se basent sur les structures spatiales comme les arbres R de [Gut84]. Un exemple d'index de la première catégorie est présenté en 8.2.1. Un rapide récapitulatif de l'utilisation des arbres R pour indexer les versions de collection est présenté en 8.2.2.

8.2.1 Clé d'index à deux dimensions

L'arbre TSB (*Time-Split B-Tree*) de [LS93] est une structure à deux dimensions utilisée dans les bases de données temporelles. Chaque nœud de l'index décrit un rectangle dans un espace temps-clé. Chaque entrée de nœud d'index est composée de triplets de la forme (*temps, clé, pointeur*). Les valeurs de *temps* et *clé* correspondent respectivement à la plus petite estampille temporelle et à la plus petite clé de la région rectangulaire de l'espace temps-clé associée au sous arbre référencé par *pointeur*. Chaque enregistrement de l'index possède une clé (non temporelle) et un intervalle de temps de transaction. L'arbre TSB regroupe les données par intervalle de temps et par valeur d'attribut. La figure 8.3 représente un arbre TSB.

L'arbre TSB est le premier index temporel qui permet des recherches à la fois sur une clé d'index non temporelle et sur le temps [Nas96, BO99].

8.2.2 Utilisation d'index spatiaux

Les index spatiaux [Kol93, ST94, BO99] comme les arbres R [Gut84] ou les variantes proposées [SRF87, BKSS90, KS91, KF94] permettent également d'indexer des données dans un espace à deux dimensions. Dans le cas temporel, par exemple, l'arbre R, ou une de ses variantes, peut

être implanté en deux ou en trois dimensions [BO99]. Lorsqu'il a deux dimensions, l'intervalle de temps associé aux données est représenté par un segment dans un espace à deux dimensions, une dimension correspond au temps, l'autre correspond à la clé d'index non temporelle. L'arbre SR (*SR-Tree*) de [KS91, Kol93] est un arbre R à deux dimensions. Lorsque qu'il a trois dimensions, l'arbre R indexe des points de la forme (*clé d'index, début de l'intervalle, fin de l'intervalle*) [BO99]. L'arbre R est également utilisée pour gérer des données bi-temporelles [BJSS98], c'est-à-dire associées à un intervalle de temps de validité et un intervalle de temps de transaction.

Pour plus de détails sur les approches basées sur les arbres R, le lecteur est invité à consulter [ST94, Nas96].

Les index de ce type ne facilitent pas les requêtes par "tranches de temps" ou par version de collection. Pour retrouver tous les éléments d'une version de collection ou d'un intervalle de temps, il faut généralement lire plusieurs nœuds de l'index. La section qui suit, présentent plusieurs structures d'index à deux dimensions qui permettent de regrouper les éléments par versions de collection en gérant des versions d'index.

8.3 Structures persistantes: gestion de versions d'index

Une structure est dite persistante [DSST89] lorsque les modifications qu'elle subit ne détruisent pas ses anciennes valeurs ; la structure possède des versions. Les auteurs de [DSST89] distinguent deux types de persistance : la persistance dite partielle (*partial persistence*), présentée en 8.3.1, et la persistance complète (*fully persistence*), présentée en 8.3.2. La première est généralement utilisée dans le cas du versionnement linéaire; toutes les versions sont accessibles, mais seule une version (la version courante) peut être mise à jour. Le deuxième type de persistance permet d'accéder et de mettre à jour n'importe quelle version.

8.3.1 Persistance partielle

La persistance partielle est principalement utilisée dans les modèles temporels où seule la version courante des données peut être mise à jour mais où toutes les versions sont accessibles en lecture. L'idée est d'associer une version de la structure d'index à chaque état de la base. Les approches qui proposent de la persistance partielle de structures d'index [Kol93, MK90, BGO+96, VV97, NS98, TML99] peuvent être classées en trois catégories : les approches sans mécanisme de partage entre les versions d'index (voir en a), les approches basées sur les mécanismes de chevauchement (voir en b) et celles qui stockent l'information sur les versions au niveau des nœuds de l'index (voir en c).

a) Versions d'index sans mécanisme de partage

Le modèle de [Kol93] gère des versions d'index de sorte qu'à une version de collection corresponde un index classique (ou monoversion). La figure 8.4 donne un exemple d'index suivant cette approche. La sémantique utilisée dans ce modèle est celle du temps. A chaque index correspond donc un intervalle de temps. Il n'y a aucun partage entre versions de collection, mais chaque version possède un pointeur vers la version précédente, le versionnement étant dans ce cas linéaire,

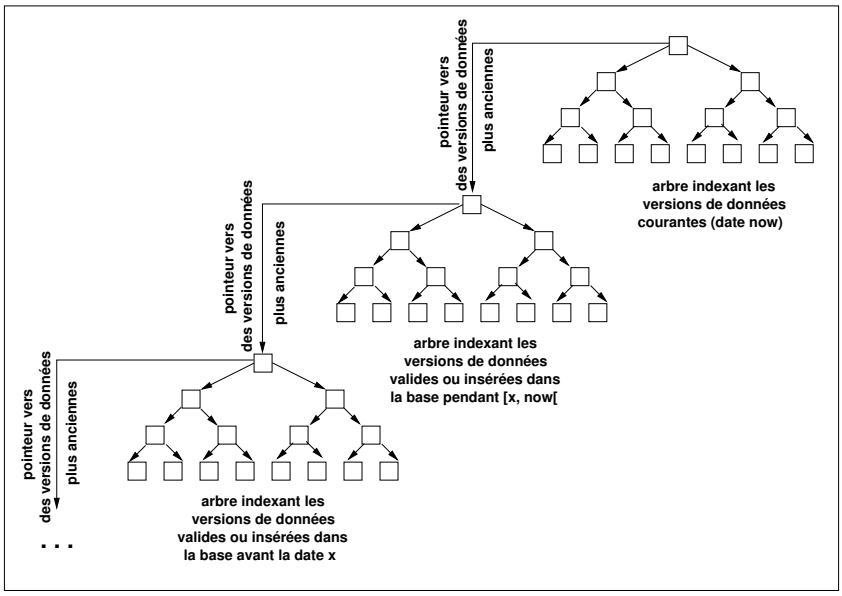

FIG. 8.4 – **La technique de versionnement d'index de [Kol93].**

puisque dépendant du temps. Par conséquent, il y a autant d'index que d'intervalles de temps gérés par la base.

Le modèle de [Kol93] se base sur deux hypothèses : [1] les requêtes accèdent plus fréquemment aux données récentes qu'aux données historiques, [2] la fréquence d'interrogation diminue avec l'ancienneté des données. Cette approche ne propose pas d'optimisation du stockage, contrairement aux approches présentées dans la section suivante.

b) Chevauchement de structures d'index

Les mécanismes de chevauchement de structures sont utilisés pour optimiser le stockage de structures persistantes. Les auteurs de [NS98] appliquent le chevauchement pour gérer un historique d'arbre R. Les mécanismes employés sont exactement ceux qui ont été présentés dans le chapitre 1 de la partie A. L'approche de [NS98] étend aux arbres R les mécanismes proposés par [MK90] pour les arbres B+.

L'approche de [TML99], également basée sur le chevauchement aux arbres B+ [MK90], ajoute un mécanisme particulier permettant de suivre l'évolution d'un nœud dans une structure chevauchée. Cette approche propose de gérer, pour chaque nœud feuille d'arbre B+, quatre pointeurs supplémentaires. Chaque nœud feuille n, dans une version v_i de l'arbre B+, possède :

1. Un pointeur $BPoint$ qui indique à partir de quel nœud, dans la version parente de v_i, le nœud n a été créé.

2. Un pointeur $BCPoint$ qui permet de lier le nœud n avec d'autres nœuds de la version v_i. Ces nœud correspondent au résultat d'un éclatement de nœud dans la version parente de v_i.

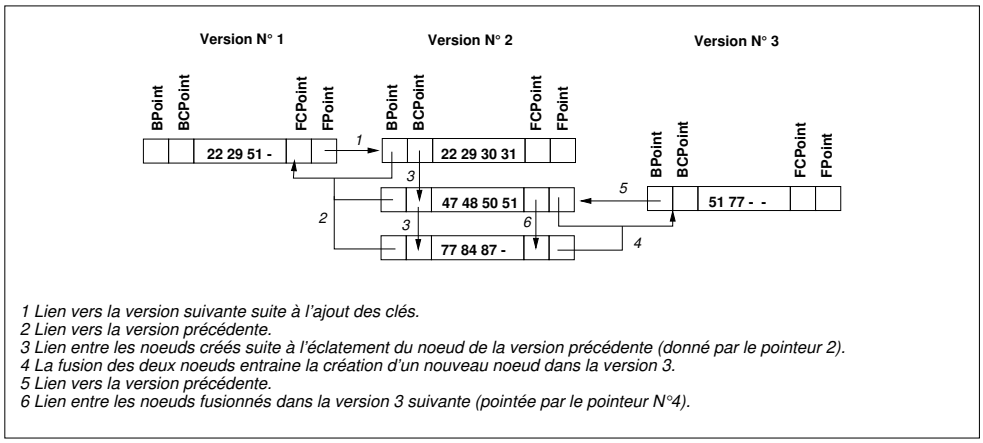

FIG. 8.5 – **Le suivi de l'évolution d'un nœud dans des arbres B+ chevauchés [TML99].**

3. Un pointeur $FCPoint$ qui lie le nœud n avec d'autres nœuds de la version v_i, ces nœuds étant fusionnés dans la version fille de v_i (le versionnement est linéaire).

4. Un pointeur $FPoint$ qui indique quel est le nœud créé à partir n, dans la version fille de v_i.

Exemple 8.4 La figure 8.5 illustre l'utilisation de ces quatre pointeurs sur un exemple de versions de nœud feuille. Dans la première version de l'arbre B+, le nœud feuille contient trois clés, 22, 29 et 51. L'ajout des clés 30, 31, 47, 48, 50, 77, 84 et 87, crée une nouvelle version d'arbre B+. Le nœud est d'abord copié dans la version 2 ; le pointeur $FPoint$ (numéroté par 1 sur la figure) référence cette copie. Puis, les insertions font éclater le nœud en trois. Le pointeur $BCPoint$, numéroté par 3 sur la figure, indique les nœuds liés par l'éclatement du nœud de la version précédente référencé par le pointeur $BPoint$ (numéroté par 2 sur la figure). Une nouvelle version 3 de l'arbre B+ est créée, dans laquelle les clés 47, 48, 50, 84 et 87 sont supprimées. Le pointeur $FCPoint$, numéroté 6 sur la figure, indique que les deux derniers nœuds de la version 2 sont fusionnés dans la version 3.

Ces quatre pointeurs ne permettent pas la gestion d'alternatives. En effet, un nœud feuille ne peut avoir qu'un seul exemplaire des quatre pointeurs supplémentaires. Ils permettent néanmoins de faire des requêtes d'historique de clé (*pure-key query*), c'est-à-dire de savoir si une clé est élément de plusieurs versions d'index [TML99], sans avoir à lire plusieurs versions de l'index. Pour permettre ces requêtes, un intervalle (*début, fin*) est associé à chaque nœud feuille. En cas de chevauchement et donc de partage, cet intervalle permet de connaître la version d'index où le nœud a été créée et à partir de quelle version d'index il n'est plus partagé.

Exemple 8.5 La figure 8.6 représente le suivi de la clé d'index 22 dans les versions 1 à 3 de l'arbre B+. Dans un premier temps la clé 22 est recherchée dans la version 1. Elle apparaît dans cette version. Puis, en suivant le pointeur $FPoint$ (en pointillé sur la figure), on atteint une feuille dans la version 2 de l'index. Cette feuille contient également la clé 22. Le pointeur $FPoint$ de la feuille, contenant la clé 22 dans la version 2 de l'index, est nul. Il faut par conséquent vérifier si ce nœud feuille est partagé par la version 3. L'intervalle associé à cette feuille est [2,3] (voir au-dessus

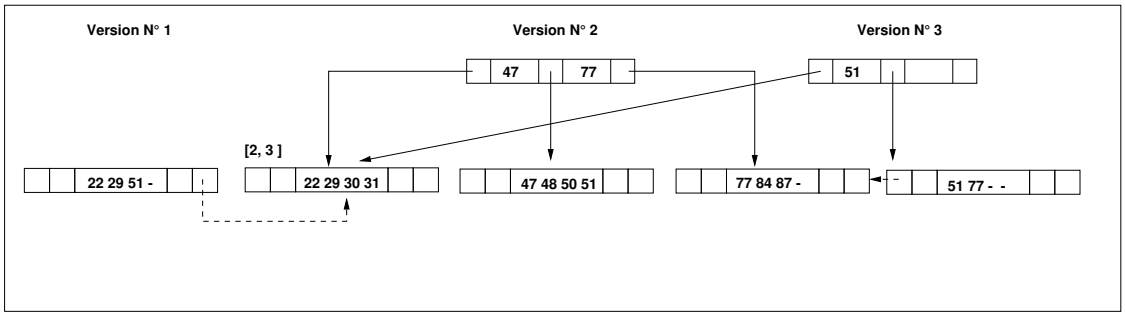

FIG. 8.6 – **Le suivi de la clé** 22 **dans une structure d'arbres B+ chevauchés.**

de la feuille dans la version 2), ce qui signifie que le nœud feuille a été créé dans la version 2 et qu'il est partagé jusqu'à la version 3. On déduit que la clé 22 appartient aux trois versions de l'index.

Dans cette approche, chaque version d'index ne contient que des clés indépendantes de la sémantique du versionnement. La sous-section suivante présente deux approches où les clés d'index de chaque version d'index contiennent de l'information sur le versionnement.

c) Stockage au niveau des nœuds de l'information sur les versions

L'arbre B Multiversion (*Multiversion B-Tree* ou MVBT) de [BGO+96] et la structure d'accès Multiversion (*Multiversion Access Structure* ou MVAS) de [VV97] proposent une autre approche de persistance partielle [ST94]. Un couple *(création, suppression)* est associé à chaque valeur de clé dans tous les nœuds de la structure. Les entités sont accessibles par une clé, par exemple leur identificateur, et l'intervalle *(création, suppression)* d'appartenance à la collection. Ces index ressemblent à l'arbre TSB de [LS93] du point de vue de sa structure et de la sémantique de versionnement, le temps. En revanche, les index de [BGO+96] et [VV97] possèdent des versions ; les valeurs de *création* et *suppression* correspondent à des identificateurs de version et non pas à des estampilles temporelles comme dans l'arbre TSB.

Chaque mise à jour (insertion ou suppression de clé) crée une nouvelle version d'index. Lorsque la racine est éclatée par une mise à jour, une nouvelle racine est créée. Une structure *root*∗ permet de connaître la racine de chaque version d'index.

Exemple 8.6 La figure 8.7 présente un exemple de structure multiversion adaptée de [BGO+96]. La première version de la structure contient les clés a, c, d, e et f. A chaque clé est associé un intervalle $[1, *[$ où $*$ signifie qu'il n'y a pas eu de suppression ; les clés ont été insérées à la version 1. Cette version est composée du nœud N_0 et de deux feuilles F_1 et F_2. La version 2 de la structure est créée par ajout de la clé b. Cet ajout modifie uniquement la feuille F_1, en ajoutant la clé b associée à l'intervalle $[2, *[$. La version 3 de la structure est créée par suppression de la clé f. Cette suppression modifie seulement la feuille F_2 en associant à la clé f l'intervalle $[1, 3]$. La version 4 de la structure est créée par l'ajout de la clé g. La feuille F_2 étant pleine, une nouvelle feuille est F_3 est créée. Les entrées de F_2 dont l'intervalle contient $*$ sont copiées dans F_3 ; il s'agit des données courantes (non supprimées avant la version 4). La clé g et son intervalle associé $[4, *[$ sont insérés

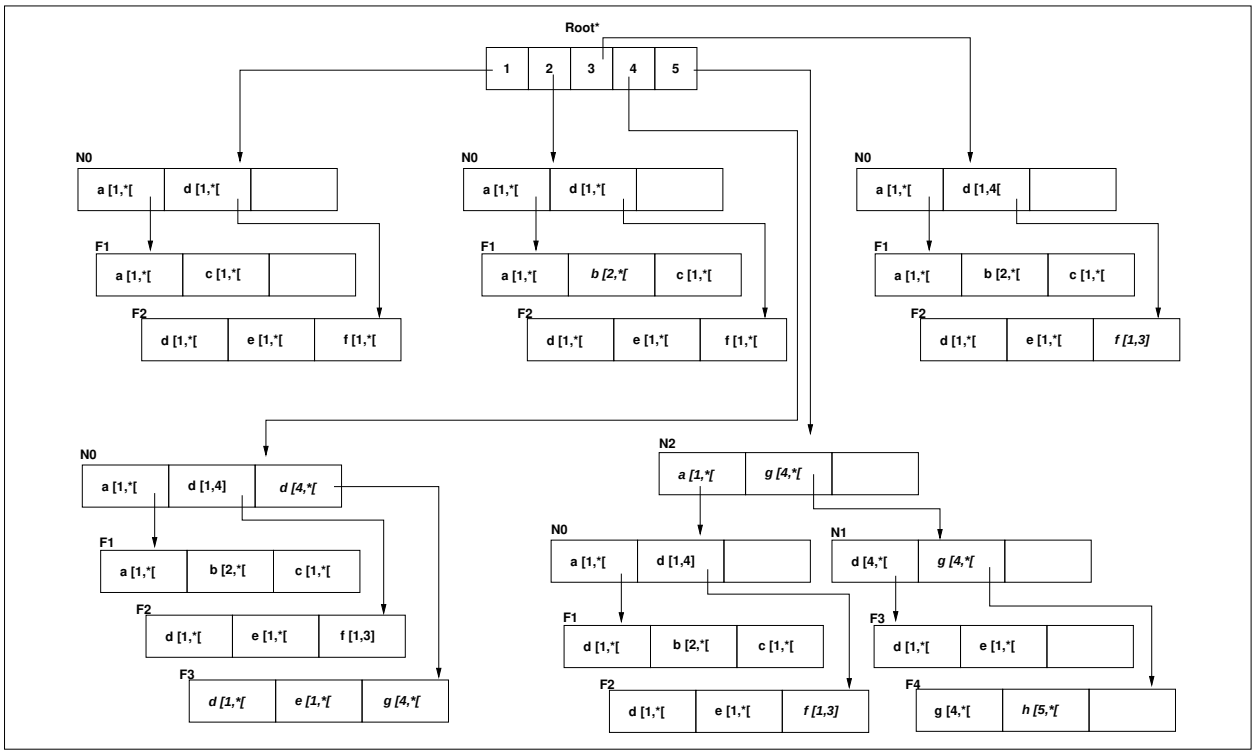

FIG. 8.7 – **Un exemple de structure multiversion adaptée de [BGO+96].**

dans F_3. La racine est mise à jour de sorte que l'entrée qui référence la feuille F_2 soit associée à l'intervalle $[1,4]$, et que l'entrée qui référence la feuille F_3 soit associée à l'intervalle $[4,*[$. La version 5 de la structure est créée par ajout de la clé h. Cet ajout éclate les nœuds jusqu'à la racine. Cette nouvelle racine est référencée par la structure $root*$ qui associe le nœud N_2 à la version 5 de l'index.

Les auteurs de [BGO+96] proposent un mécanisme de purge afin de pouvoir supprimer des versions d'index et de diminuer l'espace occupé par l'arbre B multiversion. La structure d'accès multiversion de [VV97] est basée sur les mêmes principes que [BGO+96], mais permet de partager entre versions des blocs physiques de données ayant la même valeur dans plusieurs versions successives.

8.3.2 Persistance complète

La persistance dite complète [DSST89] permet, à la différence de la persistance partielle, de mettre à jour n'importe quelle version de l'index. Chaque groupe de mises à jour se traduit par la création d'une nouvelle version d'index. Le versionnement n'est pas linéaire ; une version d'index peut avoir plusieurs successeurs. Les tableaux 8.3 et 8.4 (voir pages 154 et 155) résument les caractéristiques des différentes approches présentées dans cette section.

Les arbres B+ chevauchés du système EXODUS, présentés dans le chapitre 1 de la partie A, forment une structure persistante. D'autres modèles [LM91, Rei97] utilisent un mécanisme similaire au chevauchement, mais en permettant le partage des nœuds jusqu'à la racine. Dans ces approches, la clé d'index est indépendante de la sémantique du versionnement. Lanka et Mays [LM91] se basent sur un modèle de bases de donnée versionnée [MLDW91]. Ils proposent deux mécanismes de gestion de versions d'index pour permettre un accès rapide aux versions des différents objets de la base versionnée. La base de données est organisée en arbre B+. A une version d'arbre B+ correspond une version de la base de données. Cette approche a été étendue aux versions de collection d'objets par [Rei97]. Les différentes propositions de [LM91] sont décrites en a). L'adaptation de [Rei97] est présentée en b).

a) Méthode des gros nœuds

La première méthode proposée par [LM91] est la méthode des gros nœuds ou *Fat Node Method*. Elle se base sur les travaux de [DSST89] et consiste à conserver différentes versions d'arbre B+. Dans cette approche, un arbre B+ est composé de gros nœuds (*fat nodes*). Un gros nœud est composé d'un bloc de version (ou *Version Block*) et de plusieurs nœuds classiques (nœuds d'index ou feuilles). Un bloc de version indique le chemin à suivre dans l'arbre en fonction de l'identificateur de la version de la base. Il en résulte un arbre B+ complexe, composé de nœuds d'index et de feuilles classiques, auxquels s'ajoutent des blocs de version (voir figure 8.8).

Une lecture d'un bloc de version s'impose avant toute lecture de nœud classique. Chaque modification, ou groupe de mises à jour, sur une version non feuille (dont il existe des versions dérivées) d'arbre B+ entraîne la création d'une nouvelle version. Cette nouvelle version contient les changements incrémentaux par rapport à la version précédente. Lorsque qu'un nœud est modifié entre deux versions, une nouvelle entrée est créée dans le bloc de version correspondant au nœud. Les nœuds non modifiés entre deux versions sont partagés, via le mécanisme de bloc de version. Le partage peut être implicite. En effet, si l'identificateur de la version n'apparaît pas dans le bloc de version, alors la version partage un nœud avec son ancêtre le plus proche, dont l'identificateur apparaît dans le bloc. Ajoutée à ce mécanisme, une structure *Root** référence le nœud racine dans chaque version de l'arbre.

Exemple 8.7 La figure 8.8 donne un exemple d'application de cette méthode sur un arbre B+ possédant trois versions identifiées par 1, 2 et 3, les deux dernières dérivant de la première. Il s'agit d'une application de l'approche de [LM91] aux exemples du chapitre 7 (voir figures 7.5 et 7.6). Dans la version 2, l'enregistrement de clé 2 est ajouté dans l'arbre B+. Cet ajout modifie la feuille F_0 créant un nouveau nœud F_3 et une nouvelle entrée dans le bloc de version référençant F_0. Ce bloc indique que les enregistrements de clé inférieure à 4 sont accessibles par la feuille F_0 dans

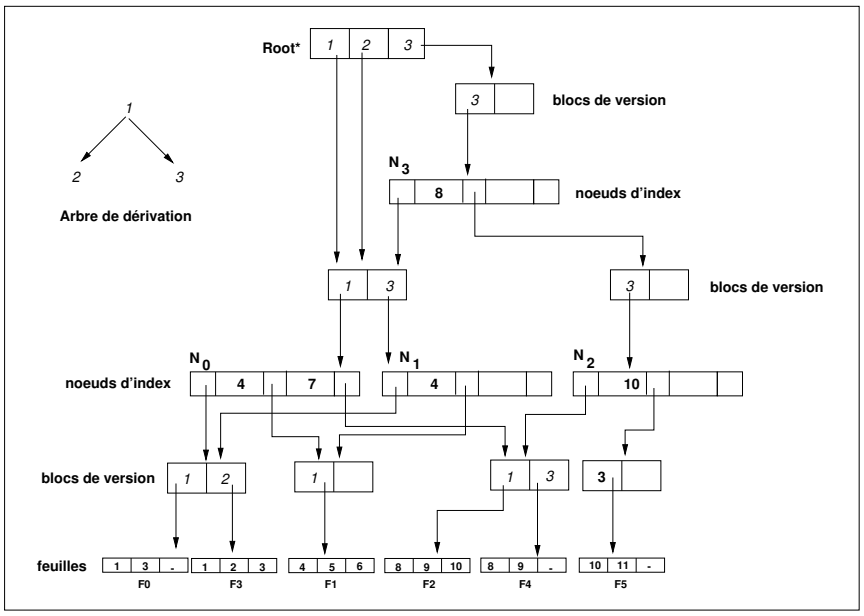

FIG. 8.8 – **Un exemple d'application de la** *Fat Node Method* **de [LM91].**

la version 1 de l'index et par la feuille F_3 dans la version 2. La version 3 de l'index est créée par ajout de la clé 11 par rapport à la version 1. Cet ajout éclate la feuille F_2 et crée deux nouvelles feuilles F_4 et F_5. La feuille F_4 est référencée par le même bloc de version que F_2. L'éclatement se poursuit jusqu'à la racine en créant les nœuds N_1 à N_3.

Il n'est pas possible, dans cette approche, de supprimer une version. Les auteurs de [LM91] n'expliquent pas cette restriction de même qu'il n'expliquent pas pourquoi la mise à jour d'une version d'index non feuille entraîne la création d'une nouvelle version d'index. A chaque identificateur de version apparaissant dans un bloc de version est associé un pointeur vers un nœud de l'index. Une suppression du partage implicite par suppression ou modification d'une version d'index v_i est susceptible de créer autant de couples $< identificateur_de_version, pointeur >$ qu'il y a de versions filles de v_i.

Afin de diminuer la place occupée par les blocs de version, Lanka et Mays [LM91] proposent une autre méthode adaptée de la première, la *Fat Field Method*. Cette méthode limite le nombre de blocs de versions. Les blocs de versions ne sont créés que pour les nœuds modifiées entre deux versions. Les autres nœuds d'index contiennent des quadruplets de la forme *(k, id_c, id_s, pointeur)*, où id_c est l'identificateur de la version de la base où l'article de clé k a été inséré, et id_s l'identificateur de la version où il a été supprimé. Il y a partage des nœuds communs entre versions, sans bloc de version intermédiaire ; les pointeurs des nœuds de l'index pointent directement vers les nœuds fils, sans référencer les blocs de versions.

Exemple 8.8 L'exemple précédent est repris en y appliquant cette méthode (voir figure 8.9) ; les flèches en pointillé représentent les liens directs (sans bloc de version) entre nœuds. Les couples (id_c, id_s) sont représentés sur les flèches associées à chaque clé. Le symbole * signifie l'infini.

Dans les deux méthodes proposées par [LM91], une mise à jour sur une version d'index non feuille créée une nouvelle version. En conséquence, le système conserve des versions d'arbre B+ dont l'utilisateur n'a peut être plus besoin, puisque la suppression n'est pas autorisée. Il y a donc stockage d'information inutile.

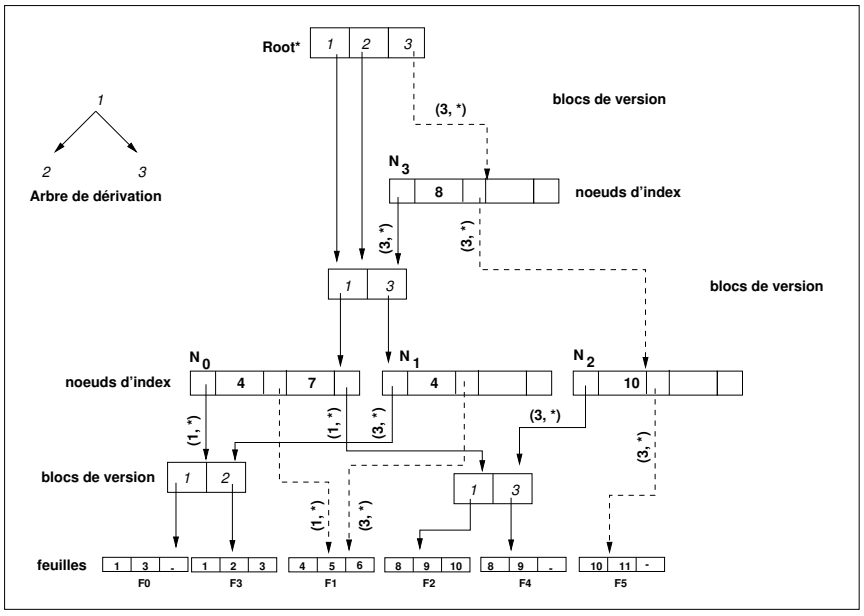

FIG. 8.9 – **Une application de la** *Fat Field Method* **de [LM91].**

b) L'adaptation de l'approche de Lanka et Mays par Reimer

Reimer [Rei97] a adapté les méthodes de [LM91], décrites précédemment, au système O_2 [BDK92] pour gérer les versions de collection d'objets. L'objectif de cette adaptation est de "supprimer" certaines contraintes de l'approche de [LM91], comme la création d'une nouvelle version à chaque mise à jour ou groupes de mises à jour sur une version non feuille de l'arbre B+, c'est-à-dire ayant des versions descendantes.

Le système O_2 gère des versions de configurations (voir chapitre 1, partie A). Afin de ne pas être obligé de créer une nouvelle configuration à chaque mise à jour (ou groupes de mises à jour) sur une version non feuille d'index, [Rei97] gère un historique de la version de collection au sein même de chaque configuration. La figure 8.10 illustre cette approche. Le mécanisme utilisé est le suivant :

- Le système O_2 gère des configurations d'objets. Un graphe de dérivation est créé pour gérer les différentes configurations de la base de données.
- Un graphe de dérivation est également géré pour chaque collection.
- Pour chaque configuration v_i, l'arbre de dérivation d'une collection donnée contient plusieurs versions $w_i^1, ...w_i^{n_i}$ associées à v_i. Seule la dernière version $w_i^{n_i}$ correspond à l'état

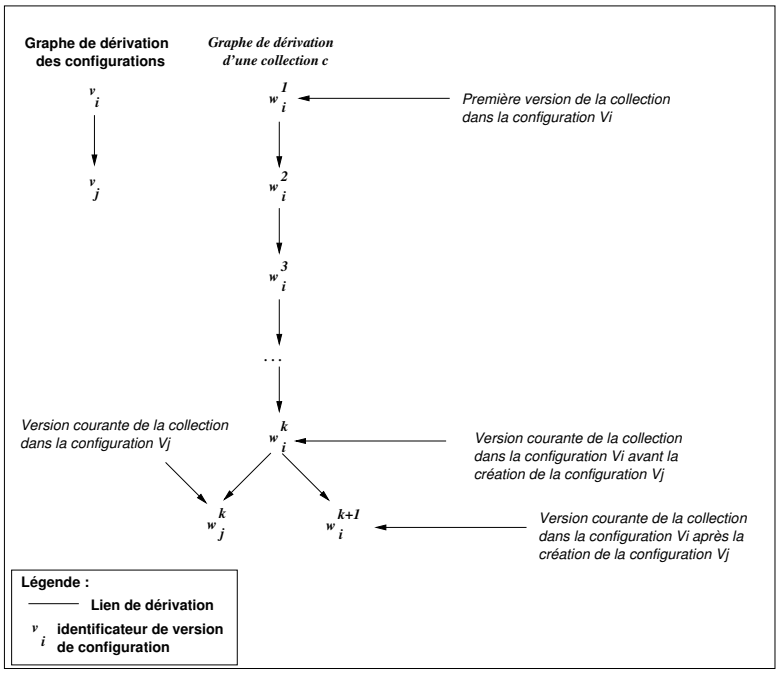

FIG. 8.10 – **L'approche de [Rei97].**

le plus récent de la collection dans la configuration v_i. Les autres versions sont des états intermédiaires.

- A chaque dérivation d'une nouvelle configuration v_j depuis v_i, deux versions de la collection, $w_j^{n_i}$ et $w_i^{n_i+1}$, sont dérivées de $w_i^{n_i}$ (état courant de la collection dans la configuration v_i). La version $w_j^{n_i}$ représente l'état initial de la collection dans la configuration v_j. La version $w_i^{n_i+1}$ représente l'état le plus récent de la collection dans la configuration v_i. Les versions dites "courantes" de la collection dans chaque configuration ne sont donc que des versions feuilles de l'arbre de dérivation de la collection.

Dans l'adaptation de Reimer, il existe, pour chaque collection, une certaine redondance d'information liée au mécanisme de gestion d'un historique propre à chaque collection au sein même des configurations de la base. En effet, lorsqu'une nouvelle configuration est dérivée, deux nouvelles versions de chaque collection sont créées, afin que l'arbre de dérivation de chaque collection ne contienne que des versions "courantes" feuilles, une version courante par configuration.

La section qui suit compare les structures d'Arbre B+ et d'Arbre R Génériques avec les approches présentées dans les sections 8.1 à 8.3.

8.4 Comparaison des Arbres B+ et R Génériques avec les autres travaux

Les objectifs de la proposition des structures d'Arbre B+ et d'Arbre R Génériques sont les suivantes :

- Gérer des versions de collection d'objets de grande cardinalité indépendamment de la sémantique du versionnement.
- Optimiser le stockage des versions de collection par partage des valeurs communes entre versions.
- Permettre aux utilisateurs de créer des versions de collection quand ils le souhaitent.
- Permettre aux utilisateurs d'appliquer toutes les opérations possibles sur les versions de collection : ajout et suppression d'objets dans une version, modification de la valeur d'un objet dans une version de collection, test d'appartenance à une version de collection, comparaison de versions, etc.

Cette section compare les Arbres B+ et R Génériques avec les autres approches en fonction des objectifs cités ci-avant. Les limites potentielles des approches sont déduites de ces objectifs, c'est-à-dire de l'utilisation des mécanismes dans un contexte parfois différent de celui pour lequel les approches ont été proposées. La sous-section 8.4.1 compare les Arbres B+ et R Génériques avec les modèles temporels. La sous-section 8.4.2 compare les Arbres B+ et R Génériques avec les modèles de versions d'index.

8.4.1 Comparaison avec les modèles temporels

Les modèles temporels [Kol93, BGO+96, VV97, NS98, TML99], étudiés précédemment, offrent, pour la plupart, une persistance partielle de l'index (voir tableau 8.1) : seule la version dite courante de l'index peut être mise à jour. Les mécanismes proposés (voir tableau 8.2) sont très dépendants de la manière avec laquelle les versions sont ordonnées. L'ordonnancement des versions est total (versionnement linéaire) et les mécanismes, basés sur cet ordonnancement, ne sont pas transposables au versionnement non linéaire, excepté pour les approches de [Kol93] et de [NS98]. En effet, les clés d'index ont, dans [EWK93, LS93, BGO+96, VV97, TML99], une composante temporelle (voir tableau 8.1). Par conséquent, les données sont organisées dans l'index en fonction des valeurs temporelles. Les auteurs de ces approches s'appuient sur le versionnement linéaire des données pour construire leur index.

La clé d'index des structures de [Kol93] et [NS98] est indépendante du temps; l'ordonnancement des versions peut par conséquent être non linéaire. A chaque intervalle de temps correspond une version de l'index. L'accès aux versions d'index est effectué en fonction d'une valeur temporelle. Mais une fois la version sélectionnée, l'index est indépendant du temps. Il est donc possible de transposer ces approches à un modèle de versions non linéaire. Les pointeurs entre versions de [Kol93] peuvent être transposés au versionnement non linéaire, chaque version d'index ayant un pointeur vers sa version parente. Toutefois, ce type d'adaptation ne peut être utilisé dans le cas où les accès aux versions d'index sont limités aux versions feuilles (sans descendant), les versions

antérieures servant uniquement d'archives et étant peu interrogées. L'approche de [NS98] limite les mises à jour des versions de collection en créant une nouvelle version d'index (et donc de collection) à chaque groupe de mise à jour. Cette limite, due aux mécanismes de chevauchement, ne permet d'utiliser cette approche que dans des applications où les mises à jour de versions d'index sont peu fréquentes.

Par conséquent, les modèles temporels sont difficilement comparables aux Arbres B+ et R Génériques, car ils sont très liés à l'ordonnancement des versions, qui est généralement linéaire. En revanche, il est plus facile de comparer les modèles de structures complètement persistantes avec les Arbres B+ et R Génériques, car le versionnement peut être non linéaire. La section qui suit présente cette comparaison.

8.4.2 Comparaison avec les modèles de versions d'index

Les structures complètement persistantes [CDRS89, MK90, LM91, Rei97] utilisent des mécanismes proches de ceux des Arbres B+ et R Génériques. Les limites des approches basées sur le chevauchement de structures ont été exposées dans le chapitre 1 de la partie A et dans la chapitre 6 de la partie B. Elles sont rapidement rappelées ici dans les tableaux 8.3 et 8.4.

Cette section compare plus particulièrement les Arbres B+ et R Génériques à l'approche de [LM91] (*Fat Node Method*).

a) Comparaison qualitative

Les mécanismes proposés dans [LM91] sont très semblables à ceux proposés dans le chapitre 7.4 à plusieurs exceptions près :

1. L'approche de [LM91] est basée sur une persistance dite complète de la structure d'index. Par conséquent, toutes les versions d'index peuvent être mises à jour. Cependant, seules les versions d'index feuilles, c'est-à-dire dont il n'existe aucune version dérivée, peuvent être directement mises à jour. Dès qu'une version d'index possède des versions filles, une mise à jour entraîne obligatoirement la création d'une nouvelle version. L'utilisateur se voit donc contraint de conserver des versions dont il n'a plus besoin puisque les mises à jour qu'il souhaitait ont été réalisées sur des versions dérivées. Cette limite est due au mécanisme de partage de nœuds entre versions d'index.
   En revanche, dans les structures d'Arbres B+ et R Génériques, aucune création de version d'index n'est imposée. L'utilisateur peut mettre à jour directement (sans création de version) toutes les versions d'index et peut créer, quand il le souhaite, de nouvelles versions d'index.

2. Les gros nœuds de l'approche *Fat Node Method* sont assez similaires aux nœuds génériques des Arbres B+ et R Génériques. Toutefois les gros nœuds de [LM91] ne représentent pas le partage réel entre versions d'index. La figure 8.11 montre un exemple : par le mécanisme de partage implicite, il serait possible de déduire que le nœud f_0 est partagé par les versions d'index 1 et 3 alors que ce nœud n'existe pas dans la version 3. En effet, la feuille f_0 n'est pas accessible par le nœud n_1 racine de la version 3 ; les clés 1 et 3 ont été supprimées dans la version 3 de l'index par rapport à la version 1. En outre, dans un bloc de version, à chaque

TAB. 8.1 – **Récapitulatif des index temporels**

Approches	**Clé d'index**	**Persistance**
Time Index [EWK93]	temps de validité	non
TSB-Tree (*Time-Split B-Tree*) [LS93]	couple (valeur d'attribut, temps de transaction)	non
Lopsided-Index [Kol93]	valeur d'attribut non temporel pour chaque version d'index qui est associée à un intervalle de temps de transaction	partielle
Multiversion B-Tree (MVBT) [BGO+96]	couple (valeur d'attribut, temps de transaction)	partielle
Arbre R historique [NS98]	valeur d'attribut non temporel pour chaque version d'index qui est associée à un intervalle de temps de transaction	partielle
Multiversion Access Structure [VV97] (MVAS)	couple (valeur d'attribut, temps de transaction)	partielle
OB+-Tree (*Overlapping B+-Trees*) [TML99]	valeur d'attribut non temporel pour chaque version d'index accessible par un identificateur	partielle

TAB. 8.2 – **Récapitulatif des index temporels (suite)**

Approches	Description	Limites potentielles
[EWK93]	Indexation de toute la base de données par un Arbre B+. Recopie des identificateurs d'entités de feuille en feuille en fonction de l'intervalle de validité, sauf lorsque les changements sont incrémentaux	Non transposable au cas de versionnement non linéaire car trop dépendant de la sémantique temporelle
[LS93]	Association à chaque nœud de l'index d'un rectangle dans un espace temps-clé	Non transposable au cas de versionnement non linéaire et pas de regroupement par version de collection
[Kol93]	Association d'une version d'index à chaque intervalle de temps. Pointeur d'une version vers la version suivante avec comme hypothèse que la fréquence des requêtes diminue avec l'ancienneté des données	Transposable au versionnement non linéaire mais dans un cas où seules les versions feuilles sont fréquemment interrogées. Redondance d'information au niveau des versions d'index.
[BGO+96]	Association à chaque clé d'index d'un intervalle d'appartenance aux versions de l'index	Non transposable au cas de versionnement non linéaire et pas de regroupement par version de collection
[NS98]	Application des mécanismes de chevauchement aux arbres R	Transposable au cas de versionnement non linéaire mais avec création d'une nouvelle version à chaque groupe de mises à jour
[VV97]	Mêmes principes que [BGO+96] avec la possibilité de partager des valeurs	Non transposable au cas de versionnement non linéaire et pas de regroupement par version de collection
[TML99]	Chevauchement d'arbres B+ avec ajout pour chaque nœud de quatre pointeurs supplémentaires pour suivre l'historique des nœuds	Non transposable au cas de versionnement non linéaire car trop dépendant de la sémantique du temps

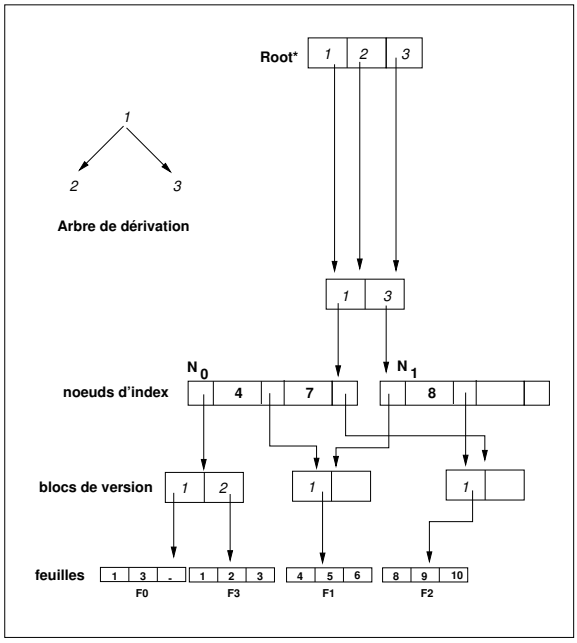

FIG. 8.11 – **Les blocs de versions n'indiquent pas le partage réel entre versions d'index.**

identificateur de version d'index est associé un pointeur vers un nœud d'index ou un nœud feuille.
En revanche, un nœud générique permet de connaître toutes les valeurs de ce nœud dans les différentes versions d'index. Un nœud générique n reflète le partage réel des valeurs de n entre versions. Une valeur spéciale, $\perp$, indique notamment l'existence ou non d'un nœud dans une version d'index. De plus, à chaque valeur de nœud est associée une liste d'identificateurs de version. La valeur d'un nœud n'est donc référencée qu'une seule fois dans le nœud générique. Un nœud générique contient par conséquent moins de pointeurs qu'un bloc de version.

3. Parce que les gros nœuds ne reflètent pas toujours le partage réel entre versions d'index, un test d'appartenance d'une clé k dans deux versions d'index impose un parcours des deux versions d'index. La comparaison de plusieurs versions nécessite également une lecture totale de toutes les versions d'index à comparer.
   En revanche, dans l'Arbre B+ Générique, chaque feuille générique connaît l'identificateur de la feuille suivante (située à droite) dans chaque version d'index. Il est par conséquent possible de comparer des versions d'index en ne lisant que les nœuds génériques feuilles.

Le tableau 8.5 rappelle les apports de l'Arbre B+ Générique par rapport aux modèles de versions d'index.

b) Comparaison quantitative

Soient $\mathcal{B}$ la taille d'une page d'arbre B+, $\mathcal{V}$ le nombre initial de versions d'arbre B+, $\mathcal{M}$ le nombre de mises à jour effectuées au total et $\mathcal{T}$ la taille moyenne d'un arbre B+. Les performances des approches de [CDRS89] et [LM91] sont adaptées de l'étude de [ST94].

Du point de vue de l'optimisation du stockage, l'Arbre B+ Générique est plus performant que les approches de [CDRS89] et [LM91]. En effet, dans l'Arbre B+ Générique, l'utilisateur choisit s'il veut ou non créer une nouvelle version d'arbre B+. Si l'utilisateur ne souhaite pas créér de nouvelle version, alors l'Arbre Générique contient toujours, après les $\mathcal{M}$ mises à jour, $\mathcal{V}$ versions d'arbres B+. En revanche, les approches de [CDRS89] et de [LM91] imposent la création d'une nouvelle version à chaque groupe de mises à jour ; après les $\mathcal{M}$ mises à jour, il y a dans [CDRS89] et [LM91] $(\mathcal{M} + \mathcal{V})$ versions d'arbres B+. Une mise à jour crée une nouvelle version et donc la copie de tout un chemin dans l'approche de chevauchement d'arbres B+ de [CDRS89]. L'espace supplémentaire occupé par une structure d'arbres B+ chevauchés est par conséquent de $\mathcal{M} * log_B(\mathcal{T})$. L'espace supplémentaire occupé par l'approche des gros nœuds de [LM91] est en $\mathcal{O}(\mathcal{M})$, car chaque mise à jour créée une nouvelle version, lorsque la version est non feuille. Il est à noter que si les mises à jour se font sur une version feuille d'arbre B+ (sans version fille) dans l'approche des gros nœuds de [LM91], les mises à jour peuvent se faire directement et l'espace occupé est, dans ce cas, équivalent à celui de l'Arbre B+ Générique.

Du point de vue de l'algorithme de recherche d'un élément dans une version v_i d'arbre B+, quelle que soit l'approche, il faut dans un premier temps rechercher la racine associée à la version v_i de l'arbre B+. La complexité de la recherche de la racine est $\mathcal{O}(\mathcal{V})$ pour l'Arbre B+ Générique. Les approches de [CDRS89] et [LM91] gèrent $\mathcal{M}$ versions supplémentaires, la complexité de la recherche de la racine est par conséquent $\mathcal{O}(\mathcal{V} + \mathcal{M})$. Une fois la racine trouvée, l'algorithme de recherche a une complexité classique dans l'approche de [CDRS89], c'est-à-dire $\mathcal{O}(log_B(\mathcal{T}))$. En revanche dans l'Arbre B+ Générique, comme dans l'approche de [LM91], la valeur de chaque nœud de l'index doit être recherchée. Par conséquent la complexité de l'algorithme est $\mathcal{O}(log_B(\mathcal{T}) * \mathcal{V})$.

Le test d'appartenance d'un élément à $\mathcal{X}$ versions d'arbres B+ ($\mathcal{X} \geq 2$) ne nécessite, dans l'Arbre B+ Générique, que la lecture d'un chemin de la racine à une feuille, puisqu'il est possible de naviguer à travers les feuilles génériques. La complexité de cette opération est $\mathcal{O}(\mathcal{X})$. En revanche, dans les approches de [CDRS89] et [LM91], l'élément doit être recherché dans les $\mathcal{X}$ versions d'arbres B+, le partage n'étant pas connu, soit une complexité de $\mathcal{O}(\mathcal{X} * log_B(\mathcal{T}))$.

Conclusion

Les Arbres B+ et R Génériques sont plus proches des structures persistantes [CDRS89, Kol93, LM91, Rei97, NS98] que des autres structures présentées dans ce chapitre. Comme les structures persistantes, ils gèrent des versions ou des états d'arbre associés à chaque version de collection. Néanmoins, les Arbres B+ et R Génériques offrent des mécanismes de gestion plus souples. En effet, l'utilisateur seul choisit la création de nouvelle version, aucune création ne lui est imposée contrairement aux structures persistantes. La redondance est par conséquent moins importante. En outre, les nœuds génériques permettent de suivre l'évolution des nœud d'arbre B+ ou R dans les différentes versions de l'arbre. Cette fonctionnalité est permise dans certaines approches, comme [TML99], mais uniquement dans le cas temporel. Les Arbres B+ et R Génériques proposent des mécanismes généraux pouvant s'adapter à tout type de modèle de versions puisqu'il sont indépendants de la sémantique du versionnement.

TAB. 8.3 – **Récapitulatif des modèles de versions d'index**

	EXODUS [CDRS89]	**Fat Node Method [LM91]**	**Adaptation de la Fat Node Method [Rei97]**
Versions d'entité gérés par la structure d'index	Objets individuels de grande taille	Base de données	Versions de collection d'objets
Description de l'approche	Chevauchement d'arbres B+	Un seul arbre B+, pour toute la base, composé de gros nœuds, c'est-à-dire d'ensembles constitués d'un bloc de version et d'un ou plusieurs nœuds classiques. Cet arbre B+ représente toutes les versions de la base	Adaptation des travaux de Lanka et Mays à O_2, avec la gestion d'un historique propre à chaque collection au sein même des configurations de la base
Lecture d'une version d'index	De complexité égale à celle d'un index monoversion modulo la recherche de la racine de la version d'index	De complexité un peu plus grande qu'un index monoversion puisque l'arbre B+ a un nombre double de niveaux par rapport à un index classique	Même complexité que [LM91]
Redondance d'information	Redondance car copie de tout un chemin depuis la racine entre 2 versions dérivées	Peu de redondance entre versions mais redondance par conservation de versions d'index éventuellement inutiles	Même redondance que [LM91] + redondance au niveau des versions de collection pour ne gérer que des versions courantes feuilles au sein des configurations

TAB. 8.4 – **Récapitulatif des modèles de versions d'index (suite)**

	EXODUS [CDRS89]	**Fat Node Method [LM91]**	**Adaptation de la Fat Node Method [Rei97]**
Mise à jour d'une version	Création d'une nouvelle version d'index + Copie de tous les nœuds allant de la racine aux feuilles modifiées	Si la version est non feuille, il y a création d'une nouvelle version d'index	Même principe que [LM91] mais dérivation de deux versions de collection à la création d'une nouvelle version de configuration
Suppression de versions d'index	Autorisée mais avec un parcours "exhaustif" de toutes les versions d'index pour ne pas supprimer les nœuds communs	Non autorisée	Basé sur [LM91] donc non autorisée.
Comparaison de plusieurs versions [a]	Impossible sans parcourir toutes les versions à comparer, car il n'y a aucun moyen de connaître par quelles versions un nœud est partagé	Même contrainte que [CDRS89] car les blocs de version ne reflètent pas toujours le partage réel	Même contrainte que [LM91]

[a] Cette opération n'est pas envisagée dans les approches. Il s'agit donc ici de suppositions par rapport aux modèles proposés.

TAB. 8.5 – **L'Arbre B+ Générique comparé aux autres approches**

Description du modèle	Gestion de versions d'arbres B+ comparable à celle de la *Fat Node Method* de [LM91]. En effet, les blocs de versions ont le même objectif que les nœuds génériques.
Redondance d'information	L'Arbre B+ Générique est moins redondant que les approches de [CDRS89, LM91, Rei97] car les mises à jour n'imposent pas la création d'une nouvelle version d'index. Les versions d'index sont créées lorsque l'utilisateur le demande. Il est également possible de supprimer n'importe quelle version d'arbre B+.
Lecture d'une version d'index	Au niveau de la complexité de la lecture d'une version d'index, l'Arbre B+ Générique est équivalente à l'approche de [LM91] et a fortiori de [Rei97]. En effet, la structure de l'index est semblable. L'Arbre B+ Générique reste toutefois un peu plus complexe en lecture que l'approche de [CDRS89], qui gère un index monoversion par version.
Mise à jour	La création d'une nouvelle version à chaque mise à jour n'est pas imposée comme dans les modèles décrits précédemment. De ce point de vue là, l'Arbre B+ Générique offre une plus grande souplesse par rapport aux modèles de [CDRS89, LM91, Rei97].
Suppression d'une version d'index	Toutes les versions d'index peuvent être supprimées. L'Arbre B+ Générique offre donc une plus grande diversité d'opérations sur les versions que les approches de [LM91, Rei97]. L'Arbre B+ Générique n'impose pas non plus un parcours exhaustif de toutes les versions de l'index pour permettre la suppression d'une version particulière. En effet, il est possible de connaître, à tout moment, par quelles versions un nœud est partagé. Pour cette opération, l'approche est donc également plus efficace que [CDRS89].
Comparaison des versions d'index	Les nœuds génériques permettent de connaître à tout moment le partage d'information entre versions d'index, ce qui permet d'optimiser les comparaisons de versions d'index. L'Arbre B+ Générique est donc, pour la comparaison de versions d'index, plus performante que les approches précédemment étudiées.

Conclusion et perspectives

Dans cette thèse, nous avons abordé les problèmes de gestion d'objets similaires de grande taille dans les bases de données. Ces objets, à cause de leur similarité, peuvent être considérés ou gérés comme des versions. Nous proposons en réponse à ces problèmes de gérer des versions de structure arborescente. Le chapitre 1 a montré que les structures arborescentes facilitent la gestion d'objets de grande taille qui sont fortement structurés (ex. collection d'objets) [DLR91] et offrent une certaine forme structuration à ceux qui le sont faiblement (ex. images ou BLOB) [CDRS89, Bil92b].

L'idée principale de cette thèse est la suivante : lorsque les objets de grande taille possèdent des versions ou ont des similarités avec d'autres objets, les Arbres Génériques (arbre B+, arbre R ou arbre quaternaire) gérent des ensembles regroupant des objets similaires de grande taille. Les mécanismes de ces structures ont des propriétés qui permettent simultanément (1) l'optimisation du stockage, (2) l'application d'opérations sur les objets, et (3) les comparaisons d'objets.

Les objets de grande taille ne sont pas gérés de manière isolée dans une base de données ; ils cohabitent avec d'autres entités de la base. Par conséquent, nous avons choisi de baser nos travaux sur l'approche des Versions de Bases de Données de [CJ90, CJ98]. Cette approche, présentée dans le chapitre 2, est un modèle unifié de versions : toutes les entités (objet, classe, contrainte, etc.) sont gérées avec les mêmes mécanismes. En outre ce modèle permet de conserver la cohérence entre les différentes versions d'entité qui vont ensemble.

Ce chapitre fait la synthèse des apports originaux et aborde les perspectives de notre travail. Les apports et les perspectives envisagés dans le domaine des images sont présentés en 1. Les apports et les perspectives de la généralisation des mécanismes aux structures arborescentes sont présentés en 2.

1 Représentation des images similaires

La partie B de cette thèse a présenté l'*Arbre Quaternaire Générique*, qui représente un ensemble d'images similaires organisées par des arbres quaternaires. Cette proposition a été réalisée dans le cadre d'une collaboration internationale pour répondre aux besoins des utilisateurs d'applications de traitement d'images. Ces besoins ont été exposés dans le chapitre 3.

Les approches, qui optimisent le stockage d'images similaires organisées en arbre quaternaire [VME93, VMK95, Lin97, TVM98], se basent en général sur des mécanismes de chevauchement

(*overlapping*). Le chapitre 4 a montré que ces approches, bien qu'efficaces pour diminuer l'espace de stockage des images sans augmenter la complexité de leur lecture, ne sont pas pensées pour l'application de traitement aux images. Elles sont principalement proposées pour gérer des séquences d'images similaires fixes comme les séquences d'images vidéo, et particulièrement des séquences vidéo d'images en noir et blanc [Lin97, TML99].

Cette section présente les apports de l'Arbre Quaternaire Générique (voir en 1.1) et les perspectives envisagées de cette structure (voir en 1.2).

1.1 Apports de l'Arbre Quaternaire Générique

Les apports de l'Arbre Quaternaire Générique, présenté dans le chapitre 5, sont les suivants :

- **Optimisation du stockage** : L'Arbre Quaternaire Générique optimise le stockage des images similaires en ne stockant qu'une seule fois dans la base les morceaux d'image communs à plusieurs images. La *Q-similarité* permet de calculer l'économie de stockage réalisée. Cette similarité correspond à une distance entre les arbres quaternaires qui codent les images. Deux images sont considérées comme similaires lorsque leurs arbres quaternaires ont un nombre relativement faible de nœuds différents. L'Arbre Quaternaire Générique n'est pas une nouvelle méthode de compression ou de compactage d'une image, mais les standards de compression d'images [Com87, AM88, Wal89, Dav98] peuvent être utilisés sur les morceaux d'images stockés dans l'arbre. Cette structure de données permet d'optimiser le stockage d'un ensemble d'images organisées en arbre quaternaire en les stockant de sorte que le partage soit maximum entre les images. L'objectif de l'Arbre Quaternaire Générique est de proposer un équilibre entre le stockage et l'application des opérations sur les images.

- **Découpage des images** : L'Arbre Quaternaire Générique n'est associé à aucun type d'image particulier, contrairement aux approches d'arbres quaternaires linéaires [KE80, Lin97] qui ne peuvent gérer que des images en noir et blanc. Le critère de découpage des images de l'Arbre Quaternaire Générique peut être aussi bien l'homogénéité de la couleur, l'homogénéité de la texture que le choix de l'utilisateur, qui souhaite appliquer des traitements sur des régions spécifiques de l'image.

- **Localisation des régions d'image** : L'Arbre Quaternaire Générique a un intérêt particulier pour les images pour lesquelles la localisation est importante. En effet, le découpage en arbre quaternaire offre un repère fixe dans l'espace représenté par le quadrant principal de l'image.

- **Opérations de traitement d'image** : L'Arbre Quaternaire Générique a été proposé pour répondre aux besoins des utilisateurs d'applications de traitement d'images. Il permet d'appliquer n'importe quelle opération de traitement d'images (segmentation, dilatation, érosion etc.) à un morceau d'image ou à une image entière, d'insérer ou supprimer n'importe quelle image de la base, et de comparer les images ou les régions d'images. L'Arbre Quaternaire Générique est, à notre connaissance, la seule structure permettant d'optimiser le stockage des images tout en laissant la possibilité aux utilisateurs d'opérer sur les images comme si elles étaient stockées indépendamment les unes des autres.

- **Similarité de surface des images** : Le chapitre 6 présente la *S-similarité*, une similarité de surface des images organisées en Arbre Quaternaire Générique. Cette similarité correspond à une similarité visuelle des images alors que la Q-similarité correspond à une similarité des arbres quaternaires. La S-similarité peut s'appliquer aussi bien aux images entières qu'à des sous-régions d'images. Elle est calculée à partir de la représentation des images en arbre quaternaire. Le degré d'exactitude de la S-similarité peut être précisé par l'utilisateur qui choisit la profondeur des arbres quaternaires à comparer pour calculer la S-similarité. Les autres mesures de similarité [LOT94, FSN+95, ABH97, DBP97, Mit97, Fal99, NBMM98, MBNW99] peuvent être intégrées au calcul de la S-similarité.

1.2 Perspectives

Les perspectives de l'Arbre Quaternaire Générique concernent plusieurs domaines principaux : le traitement d'images (voir en a et b), l'interrogation des images (voir en c et d) et les systèmes d'information géographique (voir en d et e).

a) Traçabilité des opérations de traitement d'image

Le traitement d'image est un processus itératif de "construction" ou d'amélioration de la qualité des images. La *traçabilité* d'un processus est la possibilité de mise en correspondance des étapes franchies par celui-ci à travers des traces des opérations appliquées [Zam98]. La conservation des traces des opérations est un des besoins spécifiés par les utilisateurs de l'application de traitement d'images (voir chapitre 3). Les utilisateurs souhaitent non seulement conserver les images intermédiaires résultant d'une série d'opérations de traitement, mais également connaître l'historique des opérations effectuées. L'inclusion de la traçablité dans l'Arbre Quaternaire Générique est par conséquent un point important à étudier. En outre, cette traçabilité permettrait d'associer à l'Arbre Quaternaire Générique d'autres notions de similarité d'images, comme la similarité par transformations présentée dans la sous-section suivante.

b) Similarité par transformation

Une approche de recherche d'images similaires basée sur les transformations est présentée dans [Sub98]. Le principe de cette approche est que le niveau de dissimilarité entre deux objets o_1 et o_2 est proportionnel au coût minimum de transformation de l'objet o_1 en o_2 et vice et versa. Les utilisateurs peuvent ainsi personnaliser leur notion de similarité en précisant les opérateurs de transformation qu'ils souhaitent appliquer aux images et en associant, à chaque opérateur, une fonction de coût.

Ainsi, dans une application de traitement d'images, les images stockées dans l'Arbre Quaternaire Générique correspondent aux résultats d'opérations de traitement sur une ou plusieurs images initiales. Il est donc intéressant de laisser les utilisateurs déterminer leurs propres critères de similarité en fonction des opérations appliquées aux images.

Différentes similarités (Q-similarité, S-similarité, similarité par transformation) peuvent être associées à l'Arbre Quaternaire Générique. Par conséquent, cette structure peut aider l'interrogation des images, comme le montre la sous-section suivante.

c) Intégration de l'Arbre Quaternaire Générique dans un système de recherche d'images par similarité

Le concept de *S-similarité*, présenté dans le chapitre 6, permet d'envisager l'Arbre Quaternaire Générique comme un outil à intégrer à un système de recherche d'images par le contenu. Un tel système permet de rechercher dans la base des images similaires à une image modèle, la similarité étant globale c'est-à-dire associée à l'image entière. Il est intéressant de disposer de mécanismes d'interrogation des images, qui incluent des comparaisons locales (sur des régions) entre images. Nous avons vu, dans le chapitre 6, que peu d'approches [Smi97, MBNW99, NRS99] proposent des comparaisons locales d'images. Par rapport à ces approches, l'Arbre Quaternaire Générique permet une grande flexibilité des requêtes : choix du critère de découpage et de la profondeur des arbres quaternaires à comparer, similarité de surface des régions, etc. Il serait donc intéressant d'étudier les résultats possibles et les performances de l'Arbre Quaternaire Générique dans un système de recherche d'images par similarité semblable à ceux proposés dans les autres travaux de la recherche.

d) Interrogation des images

L'Arbre Quaternaire Générique est basé sur les mécanismes de l'approche des Versions de Bases de Données [CJ90, CJ98]. Un langage formel, VQL, a été proposé [Abd97, AJ97b] pour l'interrogation des entités multiversions. Ce langage peut être étendu à l'interrogation des images stockées dans l'Arbre Quaternaire Générique. Le type de requête peut être, par exemple, "*Trouver toutes les images telles que la région 01 a valeur v*", "*Trouver toutes les images dont la région* 002 *a même valeur*", etc.

En outre, l'arbre quaternaire est une structure offrant un repère spatial fixe dans les images. L'Arbre Quaternaire Générique a un intérêt pour les domaines où la localisation des objets dans les images est importante. Dans le domaine géographique, des images satellites sont associées aux données spatiales de la base [Sub98, SZB99]. L'Arbre Quaternaire Générique peut être envisagé pour suivre l'évolution d'une région (vérifier si elle est la même ou pas) afin de voir, par exemple, les différents types d'occupation d'un sol ou d'analyser le mitage (agrandissement des zones d'habitation) d'une région.

Il est courant d'associer de la sémantique à des photographies aériennes ou satellites, en choisissant dans l'image des objets d'intérêt. L'utilisation de l'arbre quaternaire R, présenté dans le chapitre 4, permet d'indexer les objets contenus dans des images organisées en arbre quaternaire. Un Arbre Quaternaire Générique R pourrait permettre d'interroger les images sur la sémantique des objets qu'elles contiennent et permettre par exemple de poser des requêtes telles que "*Trouver les images où un objet x apparaît dans une région donnée*".

e) Créations d'animations temporelles

La création d'animation est une manière de visualiser des phénomènes spatio-temporels (ex. le déplacement d'un cyclone, l'avancée du désert) [Hun99, HJP99]. Il est possible de créer des animations en utilisant des séries temporelles d'images fixes. L'Arbre Quaternaire Générique peut servir à la création de ce type d'animations, en affichant périodiquement les petits morceaux

différents entre les images de la série.

2 Structures Arborescentes Génériques

Les mécanismes proposés dans cette thèse ne s'appliquent pas qu'aux arbres quaternaires mais à tout type de structure arborescente. La partie C présente une généralisation des mécanismes aux arbres B+ et aux arbres R, qui sont des structures très utilisées pour l'indexation des bases de données.

Les objets de grande taille faiblement structurés comme les BLOB ou les objets fortement structurés comme les collections d'objets peuvent être gérés par des arbres B+ [CDRS89, DLR91, Bil92b]. Plusieurs approches [DSST89, CDRS89, MK90, LM91, Kol93, Rei97, TML99] proposent de gérer des structures persistantes pour indexer les objets de grande taille faiblement structurés ou les collections d'objets qui possèdent des versions. Ces structures persistantes, présentées dans le chapitre 8, gèrent des versions d'index. Cependant, à notre connaissance, aucune approche n'offre autant de fonctionnalités et de souplesse que les Arbres B+ et R Génériques.

Cette section présente les apports des Arbres B+ et R Génériques (voir en 2.1) et présente les perspectives envisagées pour ces structures (voir en 2.2).

2.1 Apports

Les apports de des Arbres B+ et R Génériques, présentés dans le chapitre 7, sont les suivants :

- **Optimisation du stockage** : Les Arbres B+ et R Génériques permettent de limiter la redondance d'information entre structures persistantes. Les nœuds communs à plusieurs arbres ne sont stockés qu'une seule fois dans la base de données. Les approches [CDRS89, LM91], présentées dans le chapitre 8, obligent l'utilisateur à conserver un nombre de versions inutiles, chaque groupe de mises à jour imposant la création d'une nouvelle version d'index. L'utilisateur des Arbres B+ et R Génériques, en revanche, gère lui-même la création des versions d'index ; aucune création de version ne lui est imposée.

- **Opérations sur les versions d'index** : Les Arbres B+ et R Génériques permettent d'appliquer, sur une version d'index, toutes les opérations réalisables sur un index classique. L'utilisateur peut insérer ou supprimer un élément dans n'importe quelle version de l'index. Il peut également insérer ou supprimer plusieurs versions d'index dans les Arbres Génériques, et les comparer. Ces opérations sont réalisées sans imposer de création supplémentaire de version comme dans [CDRS89, LM91], sans copie de chemin dans une version d'index comme dans [CDRS89, MK90, TML99], ni parcours exhaustif des différentes versions d'index comme dans [CDRS89].

- **Suivi de l'évolution** : Par le mécanisme de partage explicite et implicite, utilisé dans les Arbres Génériques, il est possible de suivre l'évolution d'un nœud dans plusieurs versions d'index, c'est-à-dire connaître la valeur d'un nœud dans différentes versions. Ce type d'opération est proposée dans l'approche de [TML99] mais uniquement dans le cas temporel. Les Arbres Génériques sont, quant à eux, indépendants de la sémantique de versionnement.

2.2 Perspectives

Les perspectives des Arbres Génériques concernent la similarité des objets indexés (voir en a), la gestion de versions de document (voir en b), le suivi de l'évolution spatio-temporelle (voir en c) ou l'indexation de données stockées dans un entrepôt de données (voir en d).

a) Traçabilité et similarité par transformation

Comme l'Arbre Quaternaire Générique, des mécanismes de gestion de la traçabilité [Zam98, ZJ99] peuvent être associés aux Arbres B+ et R Génériques. En revanche, à l'inverse des arbres quaternaires, la comparaison des arbres B+ et des arbres R est complexe car leur structure peut varier ; deux versions d'arbre peuvent avoir un nombre de niveaux ou de nœuds différents ou encore une racine différente. Par conséquent la similarité d'objets organisés en arbre B+ comme les BLOB [CDRS89, Bil92b] est difficile à envisager. En revanche, la similarité par transformation, présentée dans [Sub98], peut s'appliquer aux Arbres B+ et R Génériques, comme elle peut s'appliquer à l'Arbre Quaternaire Générique. Il est par conséquent possible de calculer une similarité de tous les objets gérés par un Arbre Générique.

b) Versions de document

La gestion de documents textuels ou hypertextuels nécessite la gestion de versions de document [Haa94, VD95, BLPS96]. Certains documents XML sont gérés à l'aide de BLOB [Mic98]. Les BLOB et les champs longs sont gérés dans les bases de données à l'aide de structures arborescentes comme les arbres B+ [DLR91]. Par conséquent, l'Arbre B+ Générique peut être envisagé pour gérer de versions de document. Cependant, une mise à jour sur un BLOB peut complètement modifier la structure de l'arbre qui l'organiser. Pour que l'approche soit intéressante, il faut donc réflechir à des mécanismes qui tiennent compte de la modification éventuelle de la structure de l'arbre.

En outre, il est possible de dériver une structure arborescente [BLMM99] des documents au format SGML [ISO86] ou XML [BPSM98, Mic98]. L'indexation des documents manipulés sur le Web étant un sujet actuel de recherche [BO99], il serait intéressant de voir si les mécanismes des Arbres Génériques peuvent être étendus à ce type de documents semi-structurés.

c) Indexation des données géographiques

Les Systèmes d'Information Géographique sont utilisés dans des domaines d'activité où la création et la gestion de scénarii alternatifs comme le suivi de l'évolution spatio-temporelle des données sont des fonctionnalités importantes [BMJ94]. L'approche des Versions de Bases de Données [CJ90] semble être une solution appropriée au problème de gestion d'objets spatio-temporels [BMJ93]. L'arbre R [Gut84] est une structure d'index spatial très utilisée dans les Systèmes d'Information Géographique [SVP+96]. Par conséquent, l'Arbre R Générique peut être envisagé pour indexer des données spatio-temporelles et faciliter leur interrogation lorsque ces données possèdent des versions.

d) Indexation des entrepôts de données

L'émergence des entrepôts de données (*datawarehouse*) a joué et joue un rôle important dans l'extension et le développement de nouvelles structures d'index [BO99]. Les structures actuellement proposées dans les travaux de recherche utilisent les mécanismes des arbres B+ ou des arbres R [JDO99]. Par conséquent, il serait intéressant de voir si les Arbres Génériques permettent de faciliter l'interrogation et la gestion des entrepôts de données.

3 Conclusion générale de la thèse

Les différentes perspectives envisageables pour les Arbres Génériques (arbre B+, arbre R ou arbre quaternaire) montrent le coté général des mécanismes proposés. Dans les bases de données classiques, relationnelles ou orientées-objet, les Arbres B+ et R Génériques ont pour objectif de faciliter l'indexation des grandes volumes de versions de données fortement structurées telles que des relations contenant un grand nombre de versions de n-uplets [Abd97] ou les collections d'objets de grande cardinalité [Rei97]. Les mécanismes des Arbres Génériques s'appliquent également à la gestion aux objets de grande taille faiblement structurés, dont les images sont un exemple concret. Les structures de données arborescentes étant largement utilisées en informatique et pour des données persistantes, le champ d'applications pouvant être touché par le concept d'Arbre Générique apparaît considérable.

Bibliographie

[Abd97] T. Abdessalem. Approche des versions de bases de données: représentation et interrogation des versions. Thèse de Doctorat en Informatique, Université Paris IX Dauphine (France), juillet 1997.

[ABH97] P.M.G. Apers, H.M. Blanken, et M.A.W. Houstma. *Multimedia Databases in perspective*. Springer, 1997. ISBN: 3-540-76109-8.

[Abi97] S. Abiteboul. Querying Semi-Structured Data. Dans *Proc. of 6th Int. Conf. on Database Theory (ICDT'97)*, pages 1–18, Delphi, Grèce, 1997.

[ABQ86] M. Adiba et N. Bui Quang. Historical Multi-Media Databases. Dans *Proc. of the 12th Int. Conf. on Very Large DataBases (VLDB'86)*, Kyoto (Japan), 1986.

[ACF+96] M. Arya, W. Cody, C. Faloutsos, J. Richardson, et A. Toga. *Multimedia Databases Systems: Issues and Research Directions - V.S. Subrahmanian et S. Jajodia (Eds.)*, Chapitre Design and Implementation of QBISM, a 3D Medical Image Database System, pages 79–100. Springer, 1996.

[Adi88] M.E. Adiba. Histories and Versions for Multimedia Complex Objects. *IEEE Data Engineering Bulletin*, 11(4):3–10, déc. 1988.

[AJ89] R. Agrawal et H. Jagadish. On Correctly Configuring Versioned Objects. Dans *Proc. of Int. Conf. on Very Large Databases (VLDB'89)*, 1989.

[AJ97a] T. Abdessalem et G. Jomier. VQL: A Query Language for Multiversion Databases. Dans *Workshop on DataBase Programming Languages (DBPL'97)*, Estes Park, Colorado (USA), 18-20 août 1997.

[AJ97b] T. Abdessalem et G. Jomier. VQL: Un langage de requêtes pour bases de données multiversions. Dans *13èmes Journées de Bases de Données Avancées (BDA'97)*, Grenoble (France), 9-11 sept. 1997.

[AJFL93] L. Al-Jadir, G. Falquet, et M. Léonard. Context Versions in an Object-Oriented Model. *Lecture Notes in Computer Science (Springer Verlag)*, 720, 1993. ISSN 0302-9743.

[AM88] Aldus Corporation et Microsoft Corporation. Tag Image File Format (TIFF) Specification Revision 5.0. Rapport Technique, 1988.

[Aok98] P.M. Aoki. Generalizing "Search" in Generalized Search Trees. Dans *Proc. of 14th Int. Conf. on Data Engineering (ICDE'98)*, pages 380–389, Orlando, Floride US, fév. 1998. http://gist.cs.berkeley.edu/.

[ATF+97] M. Aritsugi, M. Tabata, H. Fukatsu, Y. Kanamori, et Y. Funyu. Manipulation of Image Objects and Their Versions under CORBA Environment. Dans *8th International Workshop on Database and Expert Systems Applications, DEXA '97*, pages 86–91, Toulouse, France, 1997. Roland Wagner, Helmut Thoma (Eds.), IEEE-CS Press.

[AY99] Y.A. Aslandogan et C.T. Yu. Techniques and Systems for Image and Video Retrieval. *IEEE Trans. on Knowledge and Data Engineering*, 11(1):56–63, janv.-fév. 1999.

[BBA91] M. Borhani, J-P.A. Barthès, et P. Anota. Versions in Object-Oriented Databases. Rapport Technique - Memo UTC/GI/DI/N 83, Université de Technologie de Compiègne, Dépt. de Génie Informatique, mars 1991.

[BBB+97] S. Berchtold, C. Böhm, B. Braunmüller, D.A. Keim, et H-P. Kriegel. Fast Parallel Similarity Search in Multimedia Databases. Dans *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'97)*, pages 1–12, 1997.

[BCR98] P. Bosc, F. Connan, et D. Rocacher. Flexible Querying in Multimedia Databases with an Object Query Language. Dans *8th Int. Conf. on Fuzzy Systems (FUZZ-IEEE'98)*, mai 1998.

[BDK92] F. Bancilhon, C. Delobel, et P. Kanellakis. *Building an Object-Oriented Database System: The story of O_2*. Morgan Kaufmann, 1992. ISBN : 1-55860-169-4.

[BF79] J.L Bentley et J.H. Friedman. Data structures for range searching. *ACM Computing Surveys*, 11(4):397–409, 1979.

[BGO+96] B. Becker, S. Gshwind, T. Olher, B. Seeger, et P. Widmayer. An asymptotically optimal multiversion B-tree. *The International Journal on Very Large Data-Bases*, 5(4):264–275, déc. 1996. Springer Verlag.

[BHK85] F.W. Burton, M.M. Huntbach, et J.Y.G. Kollias. Multiple Generation Text Files Using Overlapping Tree Structures. *The Computer Journal*, 28(4):414–416, août 1985.

[Bil92a] A. Biliris. An Efficient Database Storage Structure for Large Dynamic Objects. Dans *Proc. of IEEE 8th Conf. on Data Engineering (ICDE'92)*, pages 301–308, Phoenix, Arizona (USA), fév. 1992.

[Bil92b] A. Biliris. The Performance of Three Database Storage Structures for Managing Large Objects. Dans *Proc. of the ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'92)*, pages 276–285, San Diego CA USA, 1992. ACM.

[BJC98] M-J. Bellosta, G. Jomier, et W. Cellary. Consistent versioning of OODB schema and its extension. Dans *14èmes Journées Bases de Données Avancées (BDA'98)*, Hammamet - Tunisie, oct. 1998.

[BJP98] M. Bouzeghoub, M. Jouve, et P. Pucheral. *Les Bases de données en questions - Tome 1 - Modèles de stockage et d'accès - placement, indexation, navigation*, Chapitre 4 - Organisations arborescentes, pages 49–71. Hermes, 1998. ISBN: 2-86601-715-3.

[BJSS98] R. Bliujute, C.S. Jensen, S. Saltenis, et G. Slivinskas. R-Tree Based Indexing of Now-Relative Bitemporal Data. Dans *Proc. of the 24th Int. Conf. on Very Large DataBases (VLDB'98)*, pages 345–356, New-York, USA, 1998.

[BKK96] S. Berchtold, D.A. Keim, et H.P. Kriegel. The X-tree: An Index Structure for High-Dimensional Data. Dans *Proc. 22nd Int. Conf. on Very Large DataBases (VLDB'96)*, pages 28–39, Mumbai (Bombay - India), 1996.

[BKMK90] F.W. Burton, J.G. Kollias, D.G. Matsakis, et V.G. Kollias. Implementation of overlapping B-trees for time and space efficient representation of collections of similar files. *The Computer Journal*, 33(3):279–280, juin 1990. Short Note.

[BKSS90] N. Beckmann, H-P. Kriegel, R. Schneider, et B Seeger. The R*-tree: An efficient and robust access method for points and rectangles. Dans *Proc. of ACM SIGMOD Int. Conf. on Management of Data*, pages 322–331, 1990.

[BLMM99] E. Bruno, J. Le Maitre, et E. Murisasco. Expression, contrôle et exploitation de la navigation hypertextuelle dans le langage SgmlQL. Dans *15èmes Journées Bases de Données Avancées (BDA'99)*, pages 83–100, Bordeaux - France, 1999.

[BLPS96] J-P. Balpe, A. Lelu, F. Papy, et I. Saleh. *Techniques avancées pour l'hypertexte*. Hermès, 1996. ISBN: 2-86601-522-3.

[BM72] R. Bayer et C. McCreight. Organisation and Maintenance of Large Ordered Indexes. *Acta Informatica*, 1(3):173–189, 1972.

[BMJ93] C. Bauzer-Medeiros et G. Jomier. Managing Alternatives and Data Evolution in GIS. Dans *Proc. of ACM Workshop on Advances in GIS*, Arlington, Virginia, 1993.

[BMJ94] C. Bauzer-Medeiros et G. Jomier. Using versions in GIS. Dans *Proc. of Database and Expert Systems Applications Conference (DEXA'94)*, Athens, Greece, 1994.

[BN98] R. Burns et I. Narang. Version Management and Recoverability for Large Object Data. Dans *Proc. of the 1998 Workshop on Multimedia Database Management Systems (MMDBMS'98)*, Dayton, OH, USA, août 1998. http://www.cse.ucsc.edu/~randal/pubs.html.

[BO97] T. Bozkaya et M. Ozsoyoglu. Distance-Based Indexing for High-Dimensional Metric Spaces. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 26(2):357–368, 1997. http://erciyes.ces.cwru.edu/bozkaya/.

[BO99] E. Bertino et B.C. Ooi. The Indispensability of Dispensable Indexes. *IEEE Trans. on Knowledge and Data Engineering*, 11(1):17–27, janv.-fév. 1999.

[BPSM98] T. Bray, J. Paoli, et M. Sperberg-McQueen. eXtensible Markup Language (XML) 1.0, 1998. W3C Recommandation, http://www.w3.org/TR/REC-xml/.

[BYRN99] R. Baeza-Yates et B. Ribeiro-Neto. *Modern Information Retrieval.* Addison Wesley - ACM press, 1999. ISBN:0-201-39829-X.

[Cat97] R.G.G. Cattell. *Bases de Données Orientées Objets - 2ème Edition (traduction en français de Object Data Management 1994).* Addison Wesley, 1997. ISBN : 2-84180-056-3.

[CDRS89] M.J. Carey, D.J. DeWitt, J.E. Richardson, et E.J. Shekita. *Object-Oriented Concepts, Databases, and Applications*, Chapitre 14 - Storage Management for Objects in EXODUS, pages 341–369. W. Kim, F. H. Lochovsky (Eds.), Addison Wesley - ACM Press, 1989. ISBN: 0-201-14410-7.

[CF98] K.L. Cheung et A.W-c. Fu. Enhanced Nearest Neighbour Search on the R-tree. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 27(3):16–21, 1998.

[Chi94] T-c. Chieuh. Content-Based Image Indexing. Dans *Proc. of the 20th Int. Conf. on Very Large DataBases (VLDB'94)*, pages 582–593, Santiago, Chili, 1994.

[CITB92] W.W. Chu, I.T. Ieong, R.K. Taira, et C.M. Breant. A Temporal Evolutionary Object-Oriented Data Model and its Query Language for Medical Image Management. Dans *Proc. of the 18th Int. Conf. on Very Large DataBases (VLDB'92)*, pages 53–64, Vancouver, Canada, 1992.

[CJ90] W. Cellary et G. Jomier. Consistency of Versions in Object-Oriented Databases. Dans *Proc. 16th Int. Conf. on Very Large DataBases (VLDB'90)*, Brisbane (Australia), 1990.

[CJ98] W. Cellary et G. Jomier. The Database Version Approach. Cahier du LAMSADE, 1998.

[CJV94] W. Cellary, G. Jomier, et G. Vossen. Multiversion Object Constellations: A New Approach to Support a Designer's Database Work. *Engineering with Computers*, 10:230–244, 1994.

[CK95] S. Christodoulakis et L. Koveos. *Modern Databases Systems - The Object Model, Interoperability and Beyond*, Chapitre 16 - Multimedia Information Systems: Issues and Approaches, pages 318–337. W. Kim, Addison Wesley, ACM Press, 1995.

[Com79] D. Comer. The Ubiquitous B-Tree. *Computing Surveys*, 11(2), 1979.

[Com87] CompuServe Inc. GIF, Graphics Interchange Format, 1987.

[CPZ97] P. Ciaccia, M. Patella, et P. Zezula. M-tree: An Efficient Access Method for Similarity Search in Metric Spaces. Dans *Proc. of 23rd Int. Conf. on Very Large DataBases (VLDB'97) Conf.*, Athens Greece, 1997.

[CT91] J.P. Cheiney et A. Touir. FI-Quadtree: a New Data Structure for Content-Oriented Retrieval and Fuzzy Search. Dans *Proc. of the 2nd Symposium on Spatial Databases (SSD'91)*, pages 23–32, 1991.

[Dav98] G.M. Davis. A Wavelet-Based Analysis of Fractal Image Compression. *IEEE Transactions on Image Processing*, 7(2):141–153, fév. 1998.

[DBP97] A. Del Bimbo et P. Pala. Shape Indexing by Multi-Scale Representation. Dans *Images Databases and Multi-Media Search*, volume 8 de *Series on Software Engineering and Knowledge Engineering - A.W.M. Smeulders, R. Jain*, pages 59–74. World Scientific, 1997.

[DGJM96a] A. Doucet, S. Gançarski, G. Jomier, et S. Monties. Integrity Constraints and Versions. Dans *Proc. 6th Int. Workshop on Foundations of Models and Languages for Data and Objects "Integrity in Databases"*, Dagstuhl Castle (Germany), sept. 1996.

[DGJM96b] A. Doucet, S. Gançarski, G. Jomier, et S. Monties. Maintien de la cohérence dans une base de données multiversion (Maintaining Consistency in a Multiversion Database). Dans *Proc. 12èmes Journées Bases de Données Avancées*, Cassis (France), 1996.

[DL93] F. Dagorn et O. Lebaillif. Un guide d'utilisation de X-Window - partie Les Formats graphiques. http://www.univ-rennes1.fr/cri/documentations/Xwindow/guide/chapter2.12.html, 1993.

[DLR91] C. Delobel, C. Lécluse, et P. Richard. *Base de données : Des systèmes relationnels aux systèmes à objets*, Chapitre 12 - Gestion des données, pages 383–431. InterEditions, 1991. ISBN : 2-7296-0371-9.

[DRS80] C. R. Dyer, A. Rosenfeld, et H. Samet. Region representation: Boundary codes from quadtrees. *Communications of the ACM*, 23(3):171–179, 1980.

[DSBM97] C. Djeraba, I. Savory, M. Barere, et S. Marchand. Content Based Image Retrieval Model in an Object Oriented Database. Dans *Images Databases and Multi-Media Search*, volume 8 de *Series on Software Engineering and Knowledge Engineering - A.W.M. Smeulders, R. Jain*, pages 263–275. World Scientific, 1997.

[DSST89] J.R. Driscoll, N. Sarnak, D.D. Sleator, et R.E. Tarjan. Making Data Structures Persistent. *Journal of Computer and System Sciences*, 38:86–124, 1989.

[EF89] J. Estublier et J-M. Favre. Structuring Large Versionned Software Products. Dans *Proc of IEEE COMPSAC*, pages 404–411, 1989.

[EN89] R. Elmasri et S.B. Navathe. *Fundamentals of Databases Systems*, Chapitre 5 - Index Structures for Files, pages 101–132. The Benjamin/Cummings Pub. Comp. Inc., 1989. ISBN: 0-8053-0145-3.

[EWK93] R. Elmasri, G.T.J. Wuu, et V. Kouramajian. *Temporal Databases Theory, Design, and Implementation*, Chapitre 18 - The Time Index and the Monotonic B^+-tree, pages 433–456. Tansel, Clifford, Gadia, Jojodia, Segev, Snodgrass, 1993. The Benjamin Cummings Publishing, ISBN : 0-8053-2413-5.

[Fal96] C. Faloutsos. Indexing Multimedia Databases (Tutorial Notes). Dans *Int. Conf. on Extending DataBase Technology (EDBT'96)*, Avignon France, mars 1996.

[Fal99] C. Faloutsos. *Modern Information Retrieval*, Chapitre 12 - Multimedia IR: Indexing and Searching, pages 345–366. Baeza-Yates, R. and Ribeiro-Neto, B. - Addison Wesley - ACM press, 1999. ISBN:0-201-39829-X.

[FB74] R.A. Finkel et J.L. Bentley. Quad-Trees: A Data Structure for Retrieval on Composite Keys. *Acta Informatica*, 4:1–9, 1974.

[FSN+95] M. Flickner, H. Sawhney, W. Niblack, J. Ashley, et al. Query by Image and Video Content: The QBIC System. *Computer - IEEE Computer Society Press*, 28(9):23–32, sept. 1995. ISSN: 0018-9162 - Query By Image Content - IBM - http://wwwqbic.almaden.ibm.com.

[Gan94] S. Gançarski. Versions et bases de données; modèle formel, supports de langage et d'interface utilisateur. Thèse de Doctorat en Informatique, Université Paris XI Orsay (France), déc. 1994.

[GG98] V. Gaede et O. Günther. Survey on Multidimensional Access Methods (Revised Version). *ACM Computing Surveys*, 30(2):170–231, juin 1998. http://www.wiwi.huberlin.de/~gaede/survez.rey.ps.Z.

[GJ95] S. Gançarski et G. Jomier. Gestion des versions d'entité et de leurs contextes: analyse et prespectives. *Ingénierie des Systèmes d'Information*, 3(6):677–711, 1995.

[GJ96] S. Gançarski et G. Jomier. Vers un langage de manipulation pour bases de données multiversions (Toward a Manipulation Language for Multiversion Databases). Dans *12èmes Journées Bases de Données Avancées (BDA'96)*, Cassis - France, 1996.

[Gün98] O. Günther. *Environmental Information Systems*, Chapitre 3- Data Storage, pages 68–110. Springer, 1998. ISBN : 3-540-60926-1.

[Gob97] C. Goble. *The Handbook of Multimedia Information Management*, Chapitre 12 - Images Database Prototype, pages 365–404. Prentice Hall, 1997. ISBN: 0-13-207325-0.

[Gro84] W. Grosky. Toward a Data Model for Integrated Pictorial Databases. *Computer Vision, Graphics and Image Proc.*, 25:371–382, 1984.

[GRW97] S. Gançarski, J. Rykowski, et W. Wieczerzycki. On Object and Database Versionning in Distributed Environment. Dans *Proc. of the XIIth Int. Symposium on Computer and Information Sciences (ISCIS XII)*, Antalya (Turkey), 1997.

[GS93] H. GUNADHI et A. SEGEV. Efficient Indexing Method for Temporal Relations. *IEEE Transactions on Knowledge and Data Engineering*, 5(3):496–509, 1993.

[Gut84] A. GUTTMAN. R-trees : A Dynamic Index Structure for Spatial Searching. Dans *Proc. of ACM SIGMOD Int. Symp. on the Management of Data*, pages 45–57, 1984.

[Haa94] A. HAAKE. Under CoVer: The Implementation of a Contextual Version Server for Hypertext Applications. Dans *Proc. of the European Conference on Hypermedia Technologies (ECHT'94)*, pages 81–93, sept. 1994.

[HJP99] J.B. HUNTZINGER, G. JOMIER, et M.A. PEERBOCUS. Visualizing the evolution of spatio-temporal objects. Soumis à publication, 1999.

[HNP95] J.M. HELLERSTEIN, J.F. NAUGHTON, et A. PFEFFER. Generalized Search Trees for Database Systems. Dans *Proc. 21st Int. Conf. on Very Large DataBases (VLDB'95)*, pages 562–573, Zurich (Switzerland), 1995.

[Hun99] J.B. HUNTZINGER. Présentation de résultats de requêtes Spatio-Temporelles dans les Systèmes d'Information Géographique. Mémoire de DESS, Université Paris IX Dauphine, 1999.

[ISO86] Text and office systems-Standards Generalized Markup Language - ISO 8879, 1986.

[JCE+94] C.S. JENSEN, J. CLIFFORD, R. ELMASRI, S.K. GADIA, S. JAJODIA, et AL. A Consensus Glossary of Temporal Database Concepts. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 23(1):52–64, mars 1994.

[JDO99] C. JERMAINE, A. DATTA, et E. OMIECINSKI. A Novel Index Supporting High Volume Data Warehouse Insertions. Dans *Proc. of the 25th Int. Conf. on Very Large DataBases (VLDB'99)*, pages 235–246, Edinburgh, Scotland, sept. 1999.

[JMR99a] G. JOMIER, M. MANOUVRIER, et M. RUKOZ. Stockage et Gestion d'Images par un Arbre Quaternaire Générique. Dans *15èmes Journées Bases de Données Avancées (BDA'99)*, Bordeaux (France), 1999. Note de recherche du LAMSADE Num. 22.

[JMR+99b] G. JOMIER, M. MANOUVRIER, M. RUKOZ, J. RAMIREZ, et Y. VALERO. MIS: Un prototipo de un sistema de manipulación de imágenes similares. Dans *XXV Conferencia Latinoamericana de Informàtica (Panel'99)*, Asunción (Paraguay), août-sept. 1999.

[Kat90] R.H. KATZ. Toward a Unified Framework for Version Modeling in Engineering Databases. *ACM Computing Survey*, 22(4), déc. 1990.

[KB96] S. KHOSHAFIAN et H.B. BAKER. *Multimedia and Imaging Databases*. Morgan Kaufmann, 1996. ISBN: 1-55860-312-3.

[KBC+89] W. KIM, N. BALLOU, H-T. CHOU, J.F. GARZA, et D. WOELK. *Object-Oriented Concepts, Databases, and Applications*, Chapitre 11 - Features of the ORION Object-Oriented Database Systems, pages 251–282. W. Kim, F. H. Lochovsky (Eds.), Addison Wesley - ACM Press, 1989. ISBN: 0-201-14410-7.

[KBG89] W. Kim, E. Bertino, et J.F. Garza. Composite objects revisited. *ACM SIGMOD Record*, 18(2):337–347, juin 1989.

[KC88] W. Kim et H-T. Chou. Versions of Schema for Object - Oriented Databases. Dans *Proc. 14th Int. Conf. on Very Large DataBases (VLDB'88)*, Los Angeles (California), 1988.

[KE80] E. Kawagushi et T. Endo. On a method of binary picture representation and its application to data compression. *IEEE Transactions Pattern Anal. Mach. Intell.*, 2(1):27–35, 1980.

[KF94] I. Kamel et C. Faloutsos. Hilbert R-tree: An Improved R-tree Using Fractals. Dans *Proc. of the 20th Conf. on Very Large DataBases (VLDB'94)*, pages 500–509, Santiago - Chili, 1994.

[KGBW91] W. Kim, J.F. Garza, N. Ballou, et D. Woelk. Architecture of the ORION Next-Generation Database System. *IEEE Tansactions on Knowledge and Data Engineering*, 2(1), mars 1991.

[Kim93] W. Kim. Object-Oriented Databases Systems: Promises, Reality, and Future. Dans *Proc. of the Int. Conf. on Very Large DataBases (VLDB'93)*, Dublin (Ireland), 1993. également dans: Modern Databases Systems - The Object Model, Interoperability and Beyond. W. Kim, Addison Wesley, ACM Press. 1995, ISBN: 0-210-59098-0, pp. 255-280.

[KLCC77] A.N.C. Kang, R.C.T. Lee, Chin-Liang Chang, et Shi-Kuo Chang. Storage Reduction Through Minimal Spanning Trees and Spanning Forests. *IEEE Transactions on Computers*, c-26(5):425–435, mai 1977.

[Knu73] D.E. Knuth. *The art of computer programming. Vol. 3, sorting and searching.* Addison-Wesley, 1973. ISBN: 0-201-03803-X.

[Kol93] C.P. Kolvson. *Temporal Databases Theory, Design, and Implementation*, Chapitre 17 - Indexing Techniques for Historical Databases, pages 418–432. Tansel, Clifford, Gadia, Jojodia, Segev, Snodgrass, 1993. The Benjamin Cummings Publishing, ISBN : 0-8053-2413-5.

[Kow97] G. Kowalski. *Information Retrieval Systems - Theory and Implementation.* Kluwer Academic Publisher, 1997. ISBN:0-7923-9926-9.

[Kra97] S. Krakowiak. *Gestion physique des données.* Ecole Thématique Jeunes Chercheurs en Bases de Données, Port Camargue, 1997.

[Kru56] J.B. Kruskal. On the shortest spanning subtree of a graph and the traveling slesman problem. Dans *Proc. AMS*, volume 7, pages 45–50, 1956.

[KS91] C. Kolovson et M. Stonebraker. Segment Indexes: Dynamic Indexing Techniques for Multi-Dimensional Interval Data. Dans J. Clifford et R. King, éditeurs, *Proceedings of ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'91)*, pages 138–147, juin 1991.

[KS92] W. KÄFER et H. SCHÖNING. Mapping a version model to a complex-object data model. Dans *Proc. of IEEE Data Engineering*, pages 348–357, Tempe, Arizona (USA), fév. 1992.

[KS97] N. KATAYAMA et S. SATOH. The SR-tree: An Index Structure for High-Dimensional Nearest Neighbor Queries. Dans *Proc. ACM SIGMOD Int. Conf. on Management of Data (SIGMOD'97)*, pages 369–380, Tucson, Arizona, USA, 1997.

[KSF+96] F. KORN, N. SIDIROPOULOS, C. FALOUTSOS, E. SIEGEL, et Z. PROTOPAPAS. Fast Nearest Neighbor Search in Medical Image Databases. Dans *Proceedings of 22th Int. Conf. on Very Large DataBases (VLDB'96)*, pages 215–226, Mumbai (Bombay), India, sept. 1996.

[KTF98] A. KUMAR, V.J. TSOTRAS, et C. FALOUTSOS. Designing Access Methods for Bitemporal Databases. *IEEE Transactions on Knowledge and Data Engineering (TKDE'98)*, 10(1):1–2, 1998.

[KTKM96] S. KAWASHIMA, M. TABATA, Y. KANAMORI, et Y. MASUNAGA. Versioning Model of Image Objects for Easy Development of Image Database Applications. Dans *7th International Workshop on Database and Expert Systems Applications, DEXA '96*, pages 194–200, Zurich, Switzerland, September 9-10 1996. Roland Wagner, Helmut Thoma (Eds.), IEEE-CS Press, ISBN: 0-8186-7662-0.

[KU95] A. KELLER et J. ULLMAN. A version numbering scheme with a useful lexicographic ordering. Dans *Proc. of 11th Int. Conf. on Data Engineering (ICDE'95)*, pages 240–248, Taipei (Taiwan), 1995.

[Kum84] A. KUMAR. G-tree: A new data structure for organizing multidimensional data. *IEEE Trans. Knowledge and Data Eng.*, 6(2):341–347, 1984.

[LF91] E. LIPPE et G. FLORIJN. Implementation Techniques for Integral Version Management. Dans Pierre AMERICA, éditeur, *Proc. of the European Conference on Object-Oriented Programming (ECOOP)*, volume 512 de *Lecture Notes in Computer Science (Springer-Verlag)*, pages 342–359, 1991. ISBN 3-540-50053-7.

[LG91] D. LE GALL. MPEG: A Video Compression Standard for Multimedia Applications. Dans *Communications of the ACM*. 1991. ISO 11172 - SO (JTC1/SC2/WG11).

[LHD97] M.S. LEW, D.P. HUIJSMANS, et D. DENTENEER. Content-Based Image Retrieval: Optimal Keys, Texture, Projections, or Templates. Dans *Images Databases and Multi-Media Search*, volume 8 de *Series on Software Engineering and Knowledge Engineering - A.W.M. Smeulders, R. Jain*, pages 39–47. World Scientific, 1997.

[Lin97] Tsong-Wuu LIN. Compressed quadtree representations for storing similar images. *Image and Vision Computing*, 15:833–843, 1997.

[LK97] D. LOMBARDO et Z. KEMP. Toward a model for multimedia geographical information systems. Dans *Proc. of 4th Nat. Conf. on GIS Research (GISRUK'97)*, 1997. Egalement paru dans Innovations in GIS, Z. Kemp (Eds.), Pages 56-70, ISBN: 0-7484-0657-3.

[LLM98] M-C. Lafaye, G. Louis, et D. Mehira. Indexation par le processus de production appliquée à une base d'images de synthèse. *Networking and Information Systems Journal - Database, Internet, Multimedia, Telecoms*, 1(2-3):339–364, 1998.

[LM91] S. Lanka et E. Mays. Fully Persistent B+-trees. Dans James Clifford et Roger King, éditeurs, *Proceedings of the 1991 ACM SIGMOD International Conference on Management of Data*, pages 426–435, Denver, Colorado, 29–31 mai 1991.

[Lom98] D. Lomet. B-tree Page Size When Caching is Considered. *SIGMOD Record*, 27(3):28–32, sept. 1998.

[Loo92] M.E.S. Loomis. Object Versioning. *Journal of Object-Oriented Programming*, jan. 1992.

[LOT94] H. Lu, B-C. Ooi, et K-L. Tan. Efficient Image Retrieval by Color Contents. Dans *First Int. Conf. on Applications of Database (ADB-94)*, Vadstena (Sweden), juin 1994. Lecture Notes in Computer Sciences - 819 - Springer Verlag.

[LR95] C. Léon et M. Rukoz. Sistema Distribuido para Tratamiento de Imágenes. Dans *Memorias de la XXI Conferencia Latinoamericana de Informática (PANEL'95)*, Brasil, juil. 1995.

[LRR98] C. Léon, M. Rivas, et M. Rukoz. Una Herramienta en java par Aplicaciones Distribuidas de Tratamiento de Imágenes Biomédicas. Dans *Memorias de la XXIV Conferencia Latinoamericana de Informática (PANEL'98)*, Ecuador, oct. 1998.

[LS93] D. Lomet et B. Salzberg. *Temporal Databases Theory, Design, and Implementation*, Chapitre 16 - Transaction Time Databases, pages 388–417. Tansel, Clifford, Gadia, Jojodia, Segev, Snodgrass, 1993. The Benjamin Cummings Publishing, ISBN : 0-8053-2413-5.

[MAK97] F. Mokhatarian, S. Abbasi, et J. Kittler. Efficient and Robust Retrieval by Shape Content through Curvature Scale Space. Dans *Images Databases and Multi-Media Search*, volume 8 de *Series on Software Engineering and Knowledge Engineering - A.W.M. Smeulders, R. Jain*, pages 51–58. World Scientific, 1997.

[Mal99] S. Mallat. Représenter pour traiter les signaux. *MATAPLI - SMAI Société de Mathématiques Appliquées et Industrielle*, (57):13–24, jan. 1999.

[MBNW99] J. Malki, N. Boujemaa, C. Nastar, et A. Winter. Region Queries without Segmentation for Image Retrieval by Content. *3rd Int. Conference on Visual Information Systems (Visual'99)*, juin 1999. http://www-rocq.inria.fr/~awinter/publis.html.

[MC99] E. Marcano et L. Cabera. Implementación de las Operaciones Morfológicas Básicas sobre imágenes almacenadas en Quadtrees. Tesis de Licenciatura en Computación - Trabajo Especial de Grado, Escuela de Computación, Université Centrale du Venezuela, août 1999.

[MCD94] M.R. Mediano, M.A. Casanova, et M. Dreux. V-Tree - A Storage Method for Long Vector Data. Dans *Proc. 20th Int. Conf. on Very Large DataBases (VLDB'94)*, pages 321–330, Santiago (Chile), 1994.

[MG98] J. Martinez et S. Guillaume. Colour Image Retrieval Fitted to Classical Querying. *Networking and Information Systems Journal*, 1(2-3):251–278, 1998.

[Mic98] A. Michard. *XML - Langage et applications.* Eyrolles, 1998. ISBN: 2-212-09052-8.

[Mit97] M.M. Mitschke. *Flexible content-based image retrieval systems.* PhD thesis, INRIA Rocquencourt - Univeristé de Nuremberg, 1997. http://www-rocq.inria.fr/imedia/.

[MK90] Y. Manolopoulos et G. Kapetanakis. Overlapping B+-Trees for Temporal Data. Dans *Proc. of Jerusalem Conference on Information Technology (JCIT 90) - IEEE Computer Science Press #2078*, pages 491–499, Jerusalem (Israël), oct. 1990.

[MLDW91] E. Mays, S. Lanka, B. Dionne, et R. Weida. A Persistent Store for Large Shared Knowledge Bases. *IEEE Transactions on Knowledge and Data Engineering*, 3(1):33–41, mars 1991.

[MM99] J. Martinez et N. Mouaddib. Multimedia and Databases: A Survey. *Networking and Information Systems Journal*, 2(1):89–123, 1999.

[Mon97] S. Monties. Cohérence des bases de données multiversions. Thèse de Doctorat en Informatique, Université Paris I, oct. 1997.

[MPA96] M.K. Mandal, S. Panchanathan, et T. Aboulnasr. Choice of Wavelets for Image Compression. Dans *Lecture Notes in Computer Science*, volume 1133, pages 239–249. World Scientific, 1996.

[Mul81] J.K. Mullin. Change area B-Trees : A technique tp aid error recovery. *The Computer Jounal*, 24(4):367–372, 1981.

[Nas96] M.A. Nascimento. Efficient Indexing of Temporal Databases Via B+-trees. PhD Dissertation for the Degree of Doctor of Philosophy, School of Engineering and Applied Science - Southern Methodist University - Dallas - USA, juin 1996. http://calixto.cnptia.embrapa.br/∼mario/.

[Nas97] C. Nastar. Indexation d'Images par le Contenu : un Etat de l'Art. Dans *COmpression et REprésentation des Signaux Audiovisuels (CORESA'97)*, Issy Les Moulineaux - France, 1997. Journées CNET. http://www-rocq.inria.fr/imedia/.

[NBMM98] C. Nastar, N. Boujemaa, M. Mitschke, et C. Meilhac. Surfimage : Un Système Flexible d'Indexation et de Recherche d'Images. Dans *COmpression et REprésentation des Signaux Audiovisuels (CORESA'98)*, Lanion, France, 1998.

[NPT97] M. Nappi, G. Polese, et G. Tortora. IFS Based Indexing in Image Databases. Dans *Images Databases and Multi-Media Search*, volume 8 de *Series on Software Engineering and Knowledge Engineering - A.W.M. Smeulders, R. Jain*, pages 233–240. World Scientific, 1997.

[NRS99] A. Natsev, R. Rastogi, et K. Shim. WALRUS: A Similarity Retrieval Algorithm for Image Databases. Dans *Proceedings of the ACM SIGMOD International Conference on Management of Data (SIGMOD'99)*, pages 395–406, 1999.

[NS98] M.A. Nascimento et J.R.O. Silva. Toward Historical R-trees. Dans *Proc. of ACM Symposium on Applied Computing (SAC'98)*, pages 235–240, fév. 1998. http://www.dcc.unicamp.br/~mario/Papers/hrt-sac.ps.gz.

[NTOL97] Y. Niu, M. Tamer Özsu, et X. Li. 2D-h Trees: An Index Scheme for Content-Based Retrieval of Images in Multimedia Systems. Dans *IEEE International Conference On Intelligent Processing Systems 1997 (IEEE ICIPS'97)*, pages 1710–1715, Beijing, China, oct. 1997. http://www.cs.ualberta.ca/~database/multimedia/publications.html.

[NTOL99] Y. Niu, M. Tamer Özsu, et X. Li. 2-D-S Tree: An Index Structure for Content-Based Retrieval of Images. Dans *Proc. Multimedia Computing and Networking*, San Jose, California, 25-29 janv. 1999. http://www.cs.ualberta.ca/~database/multimedia/publications.html.

[OHE96] R. Orfali, D. Harkey, et J. Edwards. *Objets Répartis - Guide de survie*. Thomson publishing, 1996. ISBN : 284180-043-1.

[OTCH98] B.C. Ooi, K-L. Tan, T.S. Chua, et W. Hsu. Fast image retrieval using color-spatial information. *The VLDB Journal*, 7:115–128, 1998.

[OTOL+97] V. Oria, M. Tamer Özsu, L. Liu, X. Li, J.Z. Li, Y. Niu, et P. Iglinski. Modeling Images for Content-Based Queries: The DISIMA Approach. Dans *Second International Conference on Visual Information Systems*, pages 339–346, San Diego, CA, déc. 1997. http://www.cs.ualberta.ca/~database/multimedia/publications.html.

[Pal90] C. Palisser. Le Modèle de Versions du Système CHARLY. *6èmes Journées de Bases de Données Avancées (BDA'90)*, 1990.

[PF95] E.G.M. Petrakis et C. Faloutsos. Similarity Searching in Large Image Databases. Rapport Technique CS-TR-3388, University of Maryland, College Park, Department of Computer Science, 1995. ftp://ftp.cs.umd.edu/pub/papers/papers/ncstrl.umcp/CS-TR-3388/CS-TR-3388.ps.Z.

[PF97] E.G.M. Petrakis et C. Faloutsos. Similarity Searching in Medical Image Databases. *IEEE Transactions on Knowledge and Data Engineering*, 9(3), mai 1997.

[Pin] Y. Pinson. La compression d'images par ondelettes. http://www.atinternet.fr/image/faq/wavelet/.

[PRS95] J-P. Peloux, G. Reynal de St Michel, et M. Scholl. Evaluation of Spatial Indices Implemented with the O_2. *Ingénierie des Systèmes d'Information*, 3(4), 1995. ftp://sikkim.cnam.fr/pub/Reports/VersoReport-069.ps.gz.

[RC99] D. Rocacher et F. Connan. Flexible queries in object-oriented databases: on the study of bags. Dans *9th Int. Conf. on Fuzzy Systems (FUZZ-IEEE'99)*, 1999.

[Rei97] W. Reimer. Version-Based Index Management in an Object-Oriented Database System to support Software Engineering Applications. Dans *4th Inter. Conf. on Object-Oriented Information Systems (OOIS'97)*, pages 477–490, Brisbane - Australia, nov. 1997.

[RKV95] N. Roussopoulos, S. Kelly, et F. Vincent. Nearest Neighbor Queries. Dans *Proc. of the ACM SIGMOD Int. Conf. on the Management of Data*, pages 71–79, 1995.

[Rob81] J.T. Robinson. The K-D-B-tree: A search structure for large multidimensional dynamic indexes. Dans *Proc. ACM SIGMOD Int. Conf. on Management of Data*, pages 10–18, 1981.

[Roc75] M.J. Rockhind. The Source Code Control System. *IEEE Transactions on Software Engineering*, SE-1(4), Dec. 1975.

[RSZ96] E. Remias, G. Sheikholeslami, et A. Zhang. Block-Oriented Image Decomposition and Retrieval in Image Database Systems. Dans *International Workshop on Multi-media Database Management Systems*, Blue Mountain Lake, New York, août 1996.

[Sam84] H. Samet. The Quadtree and Related Hierarchical Structures. *Computing Surveys*, 16(2):187–260, 1984.

[Sam89] H. Samet. *The Design and Analysis of Spatial Data Structures.* Addison Wesley, 1989. ISBN : 0-201-50255-0.

[SC94] J.R. Smith et S-F. Chang. Quad-Tree Segmentation for Texture-Based Image Query. Dans *Proc. of 2nd Annual ACM Multimedia Conf.*, San Fransisco, CA. USA, oct. 1994. http://www.ctr.colombia.edu/~jrsmith/html/pubs/ACM-MM-94/segment_1.html.

[SC96] J.R. Smith et S-F. Chang. VisualSEEk: a fully automated content-based image query system. *ACM Multimedia '96*, nov. 1996. http://www.ctr.columbia.edu/~jrsmith/publications.html.

[SGZR97] G. Sheikholeslami, J. Guo, A. Zhang, et E. Remias. Image Decomposition and Representation in Large Image Database Systems. *The Journal of Visual Communication and Image Representation*, 8(2):167–181, juin 1997. http://www.cse.buffalo.edu/ gsesfah/pub/JVCIR-97.ps.

[SJ96] V.S. Subrahmanian et S. Jajodia. *Multimedia Databases Systems: Issues and Research Directions.* Springer, 1996. ISBN: 3-540-58710-1.

[SK90] B. Seeger et H-P. Kriegel. The buddy-tree: An efficient and robust access method for spatial data base systems. Dans *Proc. 16th Int. Conf. on Very Large DataBases (VLDB'90)*, pages 590–601, 1990.

[SM93] M. Schmitt et J. Mattioli. *Morphologie Mathématique*. Logique Mathematique et Informatique - Masson, 1993. ISBN: 2-225-84385-6.

[Smi97] J.R. Smith. *Integrated Spatial and Feature Image Systems: Retrieval, Analysis and Compression.* PhD thesis, Comlumbia University, 1997. http://www.ctr.columbia.edu/~jrsmith/publications.html.

[SO93] M. Stonebraker et M. Olson. Large Object Support in POSTGRES. Dans *International Conference on Data Engineering (ICDE'93)*, pages 355–362, 1993.

[SRF87] T. Sellis, N. Roussopoulos, et C. Faloutsos. The R+-Tree: A Dynamic Index For Multi-Dimensional Objects. Dans *Proc. of the 13th Int. Conf. on Very Large DataBases (VLDB' 87)*, pages 507–518, Brighton, 1987.

[ST94] B. Salzberg et V.J. Tsotras. A Comparison of Access for Time Evolving Data. Rapport Technique, Nothestern Univ. and Polytchnic Univ. USA, 1994. NU-CCS-94-21 ftp://ftp.ccs.neu.edu/pub/people/salzberg/tempsurvey.ps.gz.

[Ste86] I.P. Stewart. Quadtrees: Storage and Scan Conversion. *The Computer Journal*, 29(1):60–75, fév. 1986.

[Sub98] V.S. Subrahmanian. *Principle of Multimedia Database Systems.* Morgan Kaufmann, 1998. ISBN: 1-55860-466-9.

[SVP+96] M. Scholl, A. Voisard, J-P. Peloux, L. Raynal, et P. Rigaux. *SGBD Géographiques - Spécificités*, Chapitre 4 - Index Spatiaux, pages 73–108. Thomson Publishing, 1996.

[SZB99] G. Sheikholeslami, A. Zhang, et L. Bian. A Multi-Resolution Content-Based Retrieval Approach for Geographic Images. *GeoInformatica*, 3(2):109–139, juin 1999. Kluwer Academic Publishers.

[TCG+93] Tansel, Clifford, Gadia, Jajodia, Segev, et Snodgrass. *Temporal Databases Theory, Design, and Implementation.* The Benjamin Cummings, 1993. ISBN : 0-8053-2413-5.

[Tec95] O_2 Technology. O_2Version Management Reference Manual - Version 4.6. Documentation temporaire, 1995.

[Tic85] W.F. Tichy. RCS-A System for Version Control. *Software Pratice and Experience*, 15(7), Juillet 1985.

[TK91] A. Touir et B. Kerhervé. Pattern Translation in Images Encoded by Linear Quadtree. Dans *Proc. of IFIP TC 5.10 Working Conf. on Modeling in Computer Graphics*, pages 231–246, Tokyo (Japon), avr. 1991.

[TML99] T. Tzouramanis, Y. Manolopoulos, et N. Lorentzos. Overlapping B+-trees : An Implementation of Transaction Time Access Method. *Data and Knowledge Eng. Journal*, 29(3):381–404, 1999.

[TO96] G. Talens et C. Ousallah. Version d'objets pour l'ingénierie. *Technique et Science Informatiques*, 15(2):145–178, 1996.

[TOC93] G. Talens, C. Oussalah, et M.F. Colinas. Versions of Simple and Composite Objects. Dans *Proc. of the 19th Int. Conf. on Very Large DataBases (VLDB'93)*, pages 62–72, Dublin (Ireland), 1993.

[TVM98] T. Tzouramanis, M. Vassilakopoulos, et Y. Manolopoulos. Overlapping Linear Quadtrees: a Spatio-temporal Access Method. Dans *6th ACM Symposium on Geographic Information Systems (ACM GIS'98)*, Washington D.C., nov. 1998.

[VD95] F. Vitali et D.G. Durand. Using Versioning to Provide Collaboration on the WWW. Dans *Proc. of the 4th Int. Conf. on the World Wide Web Journal*, Boston (USA), déc. 1995.

[Vir] Le système Virage. http://ww.virage.com.

[VME93] M. Vassilakopoulos, Y. Manolopoulos, et K. Economou. Overlapping Quadtrees for the Representation of Similar Images. *Image and Vision Computing*, 11(5):257–262, juin 1993.

[VMK95] M. Vassilakopoulos, Y. Manolopoulos, et B. Kroell. Efficiency Analysis of Overlapped Quadtrees. *Nordic Journal on Computing*, 2(1):70–84, 1995.

[VR98] Y. Vamero et J. Ramirez. QTV : Versionamiento de Imágenes mediante Quad-Trees. Tesis de Licenciatura en Computación - Trabajo Especial de Grado, Escuela de Computación, Université Centrale du Venezuela, oct. 1998.

[VV97] P.J. Varman et R.M. Verma. An Efficient Multiversion Access Structure. *IEEE Transactions on Knowledge and Data Engineering*, 9(3):391–409, mai-juin 1997.

[Wal89] G.K. Wallace. Overview of the JPEG (ISO/CCITT) still image compression standard. Dans *Proc. SPIE's Visual Communications and Image Processing*, 1989.

[Wal91] G.K. Wallace. The JPEG Still Picture Compression Standard. *Communication of ACM*, 34(4):30–44, 1991.

[WJ96] D.A. White et R. Jian. Similarity Indexing with the SS-tree. Dans *Proc. of the 12th Int. Conf. on Data Engineering (ICDE'96)*, pages 516–523, Tokyo, fév. 1996. IEEE Computer Society.

[WL97] U.K. Wiil et J.J. Leggett. Hyperform: A Hypermedia System Development Environment. *ACM Transactions on Information Systems*, 15(1):1–31, jan. 1997.

[YM98a] C.T. Yu et W. Meng. *Principles of Database Query Processing for Advanced Applications*. Morgan Kaufmann Publishers, 1998. ISBN: 1-55860-434-0.

[YM98b] C.T. Yu et W. Meng. *Principles of Database Query Processing for Advanced Applications*, Chapitre 8 - Multidimensional Search Structures, pages 267–294. Morgan Kaufmann Publishers, 1998. ISBN: 1-55860-434-0.

[Zam98] M.M. Zamfiroiu. Contribution à la traçabilité du processus de conception en génie logiciel. Thèse de Doctorat en Informatique, Université Paris IX Dauphine (France), déc. 1998.

[ZJ99] M. Zamfiroiu et G Jomier. La traçabilité du processus de conception en génie logiciel,. Dans *INFOSID'99*, pages 75–91, La Garde, France, 1999.

Table des figures

1.1 L'augmentation de la résolution d'une image au cours de son affichage à l'écran [Pin]. 7
1.2 La représentation des objets de grande taille dans le système $EXODUS$ [CDRS89]. 9
1.3 L'approche ascendante : des versions individuelles d'entité individuelle. . 12
1.4 L'approche descendante : gestion d'ensemble de versions d'entités cohérentes entre elles. 14
1.5 Le chevauchement de structures arborescentes. 15
1.6 Le chevauchement de objets de grande taille dans $EXODUS$ [CDRS89]. . 16

2.1 Un exemple de base de données multiversion. 20
2.2 Les étapes pour obtenir la valeur d'une version logique d'objet. 21
2.3 Le niveau physique de la base de données exemple (voir figure 2.1). . . . 21
2.4 La mise à jour de l'objet *voiture* o_2 dans v_1, la création de l'objet *femme* o_7 et la suppression de l'objet *homme* o_1 dans la VBD v_2. 23

3.1 Un exemple de modification de régions d'image. 29
3.2 Un exemple de dilatation de régions d'image. 30
3.3 La différence entre les images de la figure 3.2. 30
3.4 La détection des bords sur les complémentaires des images de la figure 3.2. 31
3.5 La détection d'objets après la dilatation des images de la figure 3.4. . . . 31

4.1 Une image de taille $2^3 * 2^3$ organisée en arbre quaternaire [Sam84]. 38
4.2 Un exemple d'arbre quaternaire R de seuil égal à 4. 40
4.3 L'image de la figure 3.1 organisée en arbre quaternaire. 42
4.4 Le complémentaire d'une image organisée en arbre quaternaire. 42
4.5 L'intersection de deux images organisées en arbre quaternaire. 43
4.6 L'union de deux images organisées en arbre quaternaire. 44
4.7 Un exemple d'images. 44
4.8 Le chevauchement des arbres quaternaires de deux images. 45
4.9 La représentation des images de la figure 4.7 par un FI-Quadtree. 47

5.1 Le calcul du coefficient de Q-similarité des images a et b. 52
5.2 Les images sont organisées en Arbre d'Images. 53
5.3 L'Arbre Quaternaire Générique des images de la figure 5.2. 53

5.4 La modification de l'image b. 55
5.5 L'Arbre Quaternaire Générique après la modification de l'image b. 56
5.6 La création de l'image e comme copie de l'image a. 59
5.7 La modification de l'Arbre d'Image après insertion de l'image e. 60
5.8 Une image f et sa représentation en arbre quaternaire. 61
5.9 La modification de la base après l'insertion de l'image extérieure f. . . . 63
5.10 La réorganisation de l'Arbre d'Images et de l'Arbre Quaternaire Générique par suppression et réinsertion de l'image b. 64
5.11 La réorganisation de l'Arbre d'Images et de l'Arbre Quaternaire Générique par suppression et réinsertion de l'image d. 65
5.12 Le suivi de la région 031 dans les images stockées dans l'Arbre Quaternaire Générique de la figure 5.9. 67
5.13 La représentation de quatre images en arbre quaternaire. 70
5.14 L'Arbre Quaternaire Générique SPI des images de la figure 5.13. 71
5.15 Le graphe de partage, l'arbre couvrant maximum et les Arbres d'Images déduits de la figure 5.3.2. 72
5.16 L'organisation des images de la figure 5.13 en Arbre Quaternaire Générique. 75
5.17 Un exemple de trois images médicales en niveau de gris. 76
5.18 Un Arbre Quaternaire Générique stockant les images de la figure 5.17. . 77
5.19 L'écran général du prototype MIS. 78
5.20 Le menu général du prototype MIS. 79
5.21 Le complémentaire d'une région d'image. 79
5.22 L'union de deux images. 80

6.1 Une image. 85
6.2 Les étapes de la recherche d'images par région dans [MBNW99]. 87
6.3 L'uniformisation des arbres quaternaires de deux images dont on doit calculer le coefficient de S-similarité. 88

7.1 La structure générale d'un nœud interne d'arbre B d'ordre m. 108
7.2 Un arbre B d'ordre 3. 108
7.3 La structure générale des nœuds d'arbre B+ d'ordre m. 109
7.4 L'arbre B+ correspondant à l'arbre B de la figure 7.2. 109
7.5 Un arbre B+ avant et après l'insertion de l'enregistrement de clé 2. . . . 110
7.6 Un arbre B+ avant et après l'insertion de l'enregistrement de clé 11. . . 110
7.7 La représentation de régions rectangulaires sur un fond de carte. 111
7.8 Un arbre R. 111
7.9 L'arbre R de la figure 7.8 après insertion de l'objet 15. 113
7.10 L'arbre R de la figure 7.9 après insertion de l'objet 16. 114
7.11 L'Arbre B+ Générique contenant les arbres B+ des figures 7.5 et 7.6. . 116
7.12 L'Arbre R Générique contenant les arbres R des figures 7.8 à 7.10. . . . 118
7.13 La Hiérarchie d'Arbres B+ après la création d'un nouvel arbre B+ N°4 de même valeur que l'arbre B+ N°1. 121

7.14 L'Arbre B+ Générique après l'insertion des enregistrements de clés 2 et 15 dans les arbres B+ N°4. 122
7.15 L'Arbre B+ Générique après l'insertion de l'enregistrement de clé 11 dans l'arbre B+ N°1. 123
7.16 L'Arbre B+ Générique après l'insertion de l'enregistrement de clé 12 dans l'arbre B+ N°2. 124
7.17 L'insertion de deux objets 17 et 18 dans l'espace représenté par l'arbre R N°1. 125
7.18 L'Arbre R Générique après l'insertion de l'objet 17 dans l'arbre R N°4. . 126
7.19 L'Arbre R Générique après l'insertion de l'objet 18 dans l'arbre R N°1. . 127
7.20 Un exemple de versions de collection dans une base de données multi-version. 128

8.1 Un exemple d'index à une dimension. 135
8.2 Un exemple d'Index Temporel de [EWK93] et de sa variante. 136
8.3 Un exemple d'arbre TSB. 138
8.4 La technique de versionnement d'index de [Kol93]. 140
8.5 Le suivi de l'évolution d'un nœud dans des arbres B+ chevauchés [TML99]. 141
8.6 Le suivi de la clé 22 dans une structure d'arbres B+ chevauchés. 142
8.7 Un exemple de structure multiversion adaptée de [BGO+96]. 143
8.8 Un exemple d'application de la *Fat Node Method* de [LM91]. 145
8.9 Une application de la *Fat Field Method* de [LM91]. 146
8.10 L'approche de [Rei97]. 147
8.11 Les blocs de versions n'indiquent pas le partage réel entre versions d'index. 152

Liste des tableaux

4.1 **Codage des nœuds de quad-tree par le code CBLQ chevauché de [Lin97]** 46

5.1 **Tableau de partage d'une image i à insérer avec les images de la base** . . 60
5.2 **Tableau de partage de l'image f avec les images a, b, c et d** 62
5.3 **Tableau de partage de l'image d avec les images a, b, c** 65

6.1 **Recherche par similarité des caractéristiques d'image** 92
6.2 **Recherche d'images par similarité de régions** 93
6.3 **Récapitulatif des approches de chevauchement d'arbres quaternaires** . . . 95
6.4 **Comparaison de l'Arbre Quaternaire Générique avec le chevauchement** . 96

8.1 **Récapitulatif des index temporels** 150
8.2 **Récapitulatif des index temporels (suite)** 151
8.3 **Récapitulatif des modèles de versions d'index** 154
8.4 **Récapitulatif des modèles de versions d'index (suite)** 155
8.5 **L'Arbre B+ Générique comparé aux autres approches** 156

A1.1 **Valeurs de $n_{ii'}$, n_{ij} et $n_{i'j}$ pour $n \notin (N_i \cap N_j \cap N_{i'})$** 187

Annexe A1

Démonstration : la Q-similarité est une distance

Cette annexe démontre que la Q-similarité est une distance.

La Q-similarité de deux images i et i' a été définie, dans le chapitre 5, par la formule suivante :

$$d(i,i') = \frac{\mid S(i,i') \mid}{\mid U(i,i') \mid} = \frac{\text{Nombre de nœuds différents entre les arbres quaternaires de i et i'}}{\text{Total des identificateurs de nœud (sans doublon)}}$$

Pour montrer que la Q-similarité est une distance normalisée, il faut démontrer que :

1. $d(i,i') \in [0,1]$
2. $d(i,i) = 0$
3. $d(i,i') = d(i',i)$
4. $d(i,i') \leq d(i,j) + d(j,i')$ (inégalité triangulaire).

1 Démonstration de $d(i,i') \in [0,1]$

Le nombre de nœuds différents entre les arbres quaternaires de i et i' est inférieur au nombre total d'identificateurs de nœuds (sans doublon) dans l'union des deux arbres. On a donc $\mid S(i,i') \mid \leq \mid U(i,i') \mid$. Par conséquent, $d(i,i') \in [0,1]$.

2 Démonstration de $d(i,i) = 0$

La Q-similarité entre une image i et elle-même est égale à 0 : $d(i,i) = 0$. En effet, $\mid S(i,i) \mid$ est égal au nombre de nœuds de même identificateur et de valeur différente dans l'arbre quaternaire de i. Or, il n'existe pas de tels nœuds.

3 Démonstration de $d(i,i') = d(i',i)$

Il y a commutativité entre les arguments de la fonction de Q-similarité : $d(i,i') = d(i',i)$. En effet, la valeur de d est indépendante de l'ordre des images :
$\mid S(i,i') \mid = \mid S(i',i) \mid$ et $\mid U(i,i') \mid = \mid U(i',i) \mid$.

4 Démonstration de l'inégalité triangulaire

Soient trois images i, i' et j $(i \neq i' \neq j)$. On note :

- N_i : l'ensemble des identificateurs de nœud apparaissant dans l'arbre quaternaire de i,
- $N_{i'}$: l'ensemble des identificateurs de nœud apparaissant dans l'arbre quaternaire de i',
- N_j : l'ensemble des identicateurs de nœud apparaissant dans l'arbre quaternaire de j

Pour montrer que $\frac{|S(i,i')|}{|U(i,i')|} \leq \frac{|S(i,j)|}{|U(i,j)|} + \frac{|S(j,i')|}{|U(j,i')|}$, commençons par démontrer que l'inégalité $\mid S(i,i') \mid \leq \mid S(i,j) \mid + \mid S(j,i') \mid$ est toujours vraie.

4.1 Démonstration de l'inégalité $\mid S(i,i') \mid \leq \mid S(i,j) \mid + \mid S(j,i') \mid$

On note :

- $\mid S(i,i') \mid = \sum_{n \in (N_i \cup N_{i'})} n_{ii'}$, avec $n_{ii'} = 0$ si le nœud n a même valeur dans les arbres quaternaires des images i et i' et $n_{ii'} = 1$ sinon[1].
- De même, on note $\mid S(i,j) \mid = \sum_{n \in (N_i \cup N_j)} n_{ij}$ et $\mid S(j,i') \mid = \sum_{n \in (N_j \cup N_{i'})} n_{i'j}$.

a) Cas où $n \in (N_i \cap N_{i'} \cap N_j)$

Lorsque l'on observe ce qu'il se passe pour un nœud $n \in (N_i \cap N_{i'} \cap N_j)$, plusieurs cas se présentent :

1. Si n a même valeur dans les arbres quaternaires de i et i', alors $n_{ii'} = 0$. Dans ce cas :
   (a) Si le nœud n a la même valeur dans les arbres quaternaires de i et j, alors il a même valeur dans les arbres quaternaires des images j et i'.
   On déduit que $n_{ij} = n_{i'j} = 0$ et donc que $n_{ii'} \leq n_{ij} + n_{i'j}$ $(0 \leq 0)$.
   (b) Si le nœud n a une valeur différente dans les arbres quaternaires des images i et j, alors il a une valeur différente dans les arbres quaternaires des images j et i'. On déduit que $n_{ij} = n_{i'j} = 1$ et donc que $n_{ii'} \leq n_{ij} + n_{i'j}$ $(0 \leq 1 + 1)$.

1. Il est à noter que lorsque n n'existe pas dans un des arbres quaternaires de i et i', la valeur de n est $\perp$ signifiant *n'existe pas*.

2. Si le nœud n a une valeur différente dans les arbres quaternaires de i et i', alors $n_{ii'} = 1$. Dans ce cas :

   (a) Si le nœud n a même valeur dans l'arbre quaternaire de l'image j et dans l'arbre de l'image i (resp. i'), alors $n_{ij} = 0$ (resp. $n_{i'j} = 0$). Par conséquent, le nœud n a une valeur différente dans les arbre quaternaires des images j et i' (resp. i et j), $n_{i'j} = 1$ (resp. $n_{ij} = 1$).
   On a $n_{ii'} \leq n_{ij} + n_{i'j}$ $(1 \leq 1)$.

   (b) Si le nœud n a une valeur différente dans les trois arbres quaternaires, alors :
   $n_{ii'} = n_{ij} = n_{i'j} = 1$ et $n_{ii'} \leq n_{ij} + n_{i'j}$ $(1 \leq 1 + 1)$.

On déduit que $\forall n \in (N_i \cap N_{i'} \cap N_j) : n_{ii'} \leq n_{ij} + n_{i'j}$. En effet :

$\forall n \in (N_i \cap N_{i'} \cap N_j)$, $n_{ii'} = 0 \Longrightarrow n_{ij} + n_{i'j} = 0$ ou 2.

$\forall n \in (N_i \cap N_{i'} \cap N_j)$, $n_{ii'} = 1 \Longrightarrow n_{ij} + n_{i'j} = 1$ ou 2.

Par conséquent :

$$\sum_{n \in (N_i \cap N_j \cap N_{i'})} n_{ii'} \leq \sum_{n \in (N_i \cap N_j \cap N_{i'})} n_{ij} + \sum_{n \in (N_i \cap N_j \cap N_{i'})} n_{i'j}$$

b) Autres cas possibles pour n

La tableau A1.1 résume les autres cas possibles pour n.

c) Fin de la démonstration de $\mid S(i,i') \mid \leq \mid S(i,j) \mid + \mid S(j,i') \mid$

D'après le tableau A1.1, on déduit que :

1. $\sum_{(n \in (N_i) \setminus (N_j \cup N_{i'}))} n_{ii'} = \sum_{(n \in (N_i) \setminus (N_j \cup N_{i'}))} n_{ij}$
2. $\sum_{(n \in (N_{i'}) \setminus (N_i \cup N_j))} n_{ii'} = \sum_{(n \in (N_{i'}) \setminus (N_i \cup N_j))} n_{i'j}$
3. Aucune relation entre $n_{ii'}$, n_{ij} et $n_{i'j}$ ne peut être déduite lorsque $n \in (N_j) \setminus (N_i \cup N_{i'}))$
4. $\sum_{(n \in (N_i \cap N_{i'}) \setminus N_j)} n_{ii'} \leq \sum_{(n \in (N_i \cap N_{i'}) \setminus N_j)} n_{ij} + \sum_{(n \in (N_i \cap N_{i'}) \setminus N_j)} n_{i'j}$
5. $\sum_{(n \in (N_i \cap N_j) \setminus N_{i'})} n_{ii'} \leq \sum_{(n \in (N_i \cap N_j) \setminus N_{i'})} n_{ij} + \sum_{(n \in (N_i \cap N_j) \setminus N_{i'})} n_{i'j}$
6. $\sum_{(n \in (N_{i'} \cap N_j) \setminus N_i)} n_{ii'} \leq \sum_{(n \in (N_{i'} \cap N_j) \setminus N_i)} n_{ij} + \sum_{(n \in (N_{i'} \cap N_j) \setminus N_i)} n_{i'j}$

On a :

$$\begin{aligned} \mid S(i,j) \mid &= \textstyle\sum_{n \in (N_i \cup N_j)} n_{ij} \\ &= \textstyle\sum_{n \in (N_i \cap N_j \cap N_{i'})} n_{ij} + \sum_{(n \in (N_i) \setminus (N_j \cup N_{i'}))} n_{ij} + \sum_{(n \in (N_i \cap N_j) \setminus N_{i'})} n_{ij} \\ &\quad + \textstyle\sum_{(n \in (N_{i'} \cap N_j) \setminus N_i)} n_{ij} + \sum_{(n \in (N_i \cap N_{i'}) \setminus N_j)} n_{ij} + \sum_{(n \in (N_j) \setminus (N_i \cup N_{i'}))} n_{ij} \end{aligned}$$

$$\begin{aligned} \mid S(j,i') \mid &= \textstyle\sum_{n \in (N_j \cup N_{i'})} n_{i'j} \\ &= \textstyle\sum_{n \in (N_i \cap N_j \cap N_{i'})} n_{i'j} + \sum_{(n \in (N_{i'}) \setminus (N_j \cup N_i))} n_{i'j} + \sum_{(n \in (N_i \cap N_j) \setminus N_{i'})} n_{i'j} \\ &\quad + \textstyle\sum_{(n \in (N_{i'} \cap N_j) \setminus N_i)} n_{i'j} + \sum_{(n \in (N_i \cap N_{i'}) \setminus N_j)} n_{i'j} + \sum_{(n \in (N_j) \setminus (N_i \cup N_{i'}))} n_{i'j} \end{aligned}$$

TAB. A1.1 – **Valeurs de $n_{ii'}$, n_{ij} et $n_{i'j}$ pour $n \notin (N_i \cap N_j \cap N_{i'})$**

		$n_{ii'}$	n_{ij}	$n_{i'j}$	
[1]	$\forall n \in N_i \setminus (N_{i'} \cup N_j)$	1	1	n'existe pas	$n_{ii'} = n_{ij} = 1$ car n n'existe pas dans les arbres quaternaires de i' et de j. $n_{i'j}$ n'existe pas, car $n_{i'j}$ existe $\forall n \in N_{i'} \cup N_j$.
[2]	$\forall n \in N_{i'} \setminus (N_i \cup N_j)$	1	n'existe pas	1	$n_{ii'} = n_{i'j} = 1$ car n n'existe pas dans les arbres quaternaires de i et de j. n_{ij} n'existe pas, car n_{ij} existe $\forall n \in N_i \cup N_j$.
[3]	$\forall n \in N_j \setminus (N_i \cup N_{i'})$	n'existe pas	1	1	$n_{ij} = n_{i'j} = 1$ car n n'existe pas dans les arbres quaternaires de i et de i'. $n_{ii'}$ n'existe pas, car $n_{ii'}$ existe $\forall n \in N_i \cup N_{i'}$.
[4]	$\forall n \in (N_i \cap N_{i'}) \setminus N_j$	0 ou 1	1	1	$n_{ij} = n_{i'j} = 1$ car n n'existe pas dans l'arbre quaternaire de j.
[5]	$\forall n \in (N_i \cap N_j) \setminus N_{i'}$	1	0 ou 1	1	$n_{ii'} = n_{i'j} = 1$ car n n'existe pas dans l'arbre quaternaire de i'.
[6]	$\forall n \in (N_{i'} \cap N_j) \setminus N_i$	1	1	0 ou 1	$n_{ii'} = n_{ij} = 1$ car n n'existe pas dans l'arbre quaternaire de i.

D'où :

$$\begin{aligned}|\,S(i,j)\,|+|\,S(j,i')\,| \;=\; & \textstyle\sum_{n\in(N_i\cap N_j\cap N_{i'})}(n_{ij}+n_{i'j})+\sum_{(n\in(N_{i'})\setminus(N_j\cup N_i))}n_{i'j}+\sum_{(n\in(N_i)\setminus(N_j\cup N_{i'}))}n_{ij}\\ & +\textstyle\sum_{(n\in(N_i\cap N_j)\setminus N_{i'})}(n_{ij}+n_{i'j})+\sum_{(n\in(N_{i'}\cap N_j)\setminus N_i)}(n_{ij}+n_{i'j})\\ & +\textstyle\sum_{(n\in(N_i\cap N_{i'})\setminus N_j)}(n_{ij}+n_{i'j})+\sum_{(n\in(N_j)\setminus(N_i\cup N_{i'}))}(n_{ij}+n_{i'j})\end{aligned}$$

En enlevant $\sum_{(n\in(N_j)\setminus(N_i\cup N_{i'}))}(n_{ij}+n_{i'j})$ à droite de l'égalité, on obtient :

$$\begin{aligned}|\,S(i,j)\,|+|\,S(j,i')\,| \;\geq\; & \textstyle\sum_{n\in(N_i\cap N_j\cap N_{i'})}(n_{ij}+n_{i'j})+\sum_{(n\in(N_i)\setminus(N_j\cup N_{i'}))}n_{ij}+\sum_{(n\in(N_{i'}\cap N_j)\setminus N_i)}(n_{ij}+n_{i'j})\\ & +\textstyle\sum_{(n\in(N_i\cap N_j)\setminus N_{i'})}(n_{ij}+n_{i'j})+\sum_{(n\in(N_{i'}\cap N_j)\setminus N_i)}(n_{ij}+n_{i'j})\\ & +\textstyle\sum_{(n\in(N_i\cap N_{i'})\setminus N_j)}(n_{ij}+n_{i'j})\end{aligned}$$

D'après les sections a) et b), on déduit que :

$$\begin{aligned}|\,S(i,j)\,|+|\,S(j,i')\,| \;\geq\; & \textstyle\sum_{n\in(N_i\cap N_j\cap N_{i'})}n_{ii'}+\sum_{(n\in(N_{i'})\setminus(N_j\cup N_i))}n_{ii'}+\sum_{(n\in(N_i)\setminus(N_j\cup N_{i'}))}n_{ii'}\\ & +\textstyle\sum_{(n\in(N_i\cap N_j)\setminus N_{i'})}n_{ii'}+\sum_{(n\in(N_{i'}\cap N_j)\setminus N_i)}n_{ii'}+\sum_{(n\in(N_i\cap N_{i'})\setminus N_j)}n_{ii'}\end{aligned}$$

Or

$$\begin{aligned}|\,S(i,i')\,| = \; & \textstyle\sum_{n\in(N_i\cap N_j\cap N_{i'})}n_{ii'}+\sum_{(n\in(N_{i'})\setminus(N_j\cup N_i))}n_{ii'}+\sum_{(n\in(N_i)\setminus(N_j\cup N_{i'}))}n_{ii'}\\ & +\textstyle\sum_{(n\in(N_i\cap N_j)\setminus N_{i'})}n_{ii'}+\sum_{(n\in(N_{i'}\cap N_j)\setminus N_i)}n_{ii'}+\sum_{(n\in(N_i\cap N_{i'})\setminus N_j)}n_{ii'}\end{aligned}$$

En conclusion, l'inégalité $|\,S(i,i')\,|\leq|\,S(i,j)\,|+|\,S(j,i')\,|$ est toujours vraie.

4.2 Cas où $|\,U(i,i')\,|\geq|\,U(i,j)\,|$ et $|\,U(i,i')\,|\geq|\,U(j,i')\,|$

Dans ce cas, on a : $\frac{1}{|U(i,i')|}\leq\frac{1}{|U(i,j)|}$ et $\frac{1}{|U(i,i')|}\leq\frac{1}{|U(j,i')|}$ donc l'inégalité triangulaire est vérifiée.

4.3 Cas où $|\,U(i,j)\,|\leq|\,U(i,i')\,|\leq|\,U(j,i')\,|$

Dans ce cas, on a : $\frac{|S(i,j)|}{|U(i,j)|}+\frac{|S(j,i')|}{|U(j,i')|}\leq\frac{|S(i,j)|+|S(j,i')|}{|U(j,i')|}$, car $|\,U(i,j)\,|\leq|\,U(j,i')\,|$.
Or

$$|\,U(i,i')\,|=|\,N_i\setminus(N_j\cup N_{i'})\,|+|\,N_j\cap N_i\,|+|\,N_{i'}\,|$$

En ajoutant $|\,N_j\setminus(N_i\cup N_{i'})\,|$ de chaque coté de l'égalité on obtient :

$$\begin{aligned}&|\,U(i,i')\,|+|\,N_j\setminus(N_i\cup N_{i'})\,|=|\,N_i\setminus(N_j\cup N_{i'})\,|+|\,N_j\cap N_i\,|+|\,N_{i'}\,|+|\,N_j\setminus(N_i\cup N_{i'})\,|\\ &\Longleftrightarrow|\,U(i,i')\,|+|\,N_j\setminus(N_i\cup N_{i'})\,|=|\,U(j,i')\,|+|\,N_j\setminus(N_i\cup N_{i'})\,|\\ &\Longrightarrow|\,U(i,i')\,|+|\,N_j\setminus(N_i\cup N_{i'})\,|\geq|\,U(j,i')\,|\end{aligned}$$

On déduit que : $\frac{|S(i,j)|}{|U(i,j)|}+\frac{|S(j,i')|}{|U(j,i')|}\geq\frac{|S(i,j)|+|S(j,i')|}{|U(i,i')|+|N_j\setminus(N_i\cup N_{i'})|}$.

On a vu précédemment, dans la section 4.1 que :
$|\,S(i,j)\,|+|\,S(j,i')\,|\geq|\,S(i,i')\,|+\sum_{(n\in N_j,n\notin(N_i\cup N_{i'}))}(\beta_n+\gamma_n)$ et $\sum_{(n\in N_j,n\notin(N_i\cup N_{i'}))}(\beta_n+\gamma_n)=|\,N_j\setminus(N_i\cup N_{i'})\,|$

D'où :
$\frac{|S(i,j)|}{|U(i,j)|}+\frac{|S(j,i')|}{|U(j,i')|}\geq\frac{|S(i,i')|+|N_j\setminus(N_i\cup N_{i'})|}{|U(i,i')|+|N_j\setminus(N_i\cup N_{i'})|}\geq\frac{|S(i,i')|}{|U(i,i')|}$

La démonstration de $\frac{|S(i,i')|N_j \setminus (N_i \cup N_{i'})|}{|U(i,i')|+|N_j \setminus (N_i \cup N_{i'})|} \geq \frac{|S(i,i')|}{|U(i,i')|}$ se fait par l'absurde, car on a l'hypothèse que $\mid S(i,i') + \mid \leq \mid U(i,i') \mid$.

L'inégalité triangulaire est donc vérifiée.

4.4 Cas où $\mid U(j,i') \mid \leq \mid U(i,i') \mid \leq \mid U(i,j) \mid$

La démonstration est la même que dans la section précédente.

4.5 Cas où $\mid U(i,i') \mid \leq \mid U(i,j) \mid$ et $\mid U(i,i') \mid \leq \mid U(j,i') \mid$

La démonstration est également la même dans ce cas. En effet, il suffit de réduire $\frac{|S(i,j)|}{|U(i,j)|} + \frac{|S(j,i')|}{|U(j,i')|}$ au même dénominateur, en prenant comme hypothèse que $\mid U(j,i') \mid \leq \mid U(i,j) \mid$ ou que $\mid U(i,j) \mid \leq \mid U(j,i') \mid$.

4.6 Conclusion de la démonstration

$\forall i, i', j$ on a toujours :

$$\frac{\mid S(i,i') \mid}{\mid U(i,i') \mid} \leq \frac{\mid S(i,j) \mid}{\mid U(i,j) \mid} + \frac{\mid S(j,i') \mid}{\mid U(j,i') \mid}$$

La Q-similarité est donc bien une distance normalisée.

Annexe A2

Recherche et insertion dans un arbre B+ et dans un arbre R

Cette annexe présente les algorithmes de recherche et d'insertion dans un arbre B+ et dans un arbre R.

1 Algorithmes de recherche et d'insertion dans un arbre B+

1.1 Recherche d'un enregistrement de clé K dans un arbre B+

Cet algorithme est une adaptation de l'algorithme de recherche présenté dans [EN89].

```
DEBUT
      Soit n le nœud racine de l'arbre B+

      Lire le nœud n
      Tant que n n'est pas un nœud feuille
                Début
                Soit q, le nombre de pointeurs vers un nœud fils contenus dans n
                Si K ≤ n.K_1  * n.K_i représente la ième clé du nœud n *
                   Alors n ← n.P_1  * n.P_i représente la ième pointeur vers un nœud fils de n *
                Sinon Si K > n.K_{q-1}
                          Alors n ← n.P_q
                      Sinon
                            Début
                            Chercher le nœud n d'entrée i telle que n.K_{i-1} < K ≤ n.K_i
                            n → n.P_i
                            Fin
                Lire le nœud n
                Fin
      Chercher dans le nœud n l'entrée (K_i, Pr_i) avec K = K_i.
      Si l'entrée (K_i, Pr_i) est trouvée Chercher l'enregistrement pointé par Pr_i
      Sinon L'enregistrement de clé K n'existe pas
FIN
```

1.2 Insertion d'un enregistrement de clé K dans un arbre B+ d'ordre m

L'algorithme d'insertion qui suit est une adaptation des algorithmes d'insertion présentés dans [EN89] et [CDRS89].

```
DEBUT
    Soit n le nœud racine de l'arbre B+
    Soit  S une pile initialisée à vide  * S est utilisée en cas d'éclatement de nœud *

    Tant que n n'est pas un nœud feuille
            Début
            S ← l'adresse de n
            Soit q, le nombre de pointeurs vers un nœud fils contenus dans n
            Si K ≤ n.K1  * n.Ki représente la ième clé du nœud n *
               Alors n ← n.P1  * n.Pi représente la ième pointeur vers un nœud fils de n *
            Sinon Si K > n.Kq-1 Alors n ← n.Pq
                  Sinon
                        Début
                        Chercher le nœud n d'entrée i telle que n.Ki-1 < K ≤ n.Ki
                        n → n.Pi
                        Fin
            Lire le nœud n
            Fin

    Chercher dans le nœud n l'entrée (Ki, Pri) avec K = Ki.

    Si l'entrée (Ki, Pri) est trouvée Alors L'insertion n'a pas lieu
    Sinon
          Début1
          Créer une entrée (K, Pr) avec Pr pointant vers l'enregistrement de clé K
                  Si le nœud n est non plein Alors Insérer (K, Pr) dans n à la bonne place
                  Sinon
                        Début2
                        Soit fin_algo un booléen initialisé à faux
                        Soit P un pointeur tel que P ← Pr
                        Tant que fin_algo différent de vrai
                                  Si S non vide
                                      Début3
                                      Soit pere un nœud initialisé à vide
                                      pere ← Sommet(S)  *pere reçoit le nœud père de n *
                                      Soit q ← le nombre de pointeurs vers un nœud fils de pere
                                      Soit f un nœud initialisé à vide  * f est un nœud frère de n *
                                      Soit i l'indice du pointeur pere.Pi tel que Pi pointe sur n
                                      * Si n est le premier fils de pere alors f reçoit le frère droit de n *
                                      Si i = 1 Alors f ← le nœud pointé par pere.Pi+1
                                      * Si n est le dernier fils de pere alors f reçoit le frère gauche de n *
                                      Sinon Si i = q Alors f ← le nœud pointé par pere.Pi-1
                                              * Si n est n'est ni le premier ni le dernier fils de pere *
                                            Sinon
                                                  Début
                                                  * f reçoit le frère gauche de n *
                                                  f ← le nœud pointé par pere.Pi-1
                                                  * Si f est plein alors f reçoit le frère droit de n *
```

```
        Si f est plein
            Alors f ← le nœud pointé par pere.P_{i+1}
        Fin
* Si un des frères de n est non plein, *
* Alors il y a répartition des clés *
Si (f diffèrent de nil) et (f est non plein) Alors
    Début_4
    Copier n et f dans temp en ordonnant les entrées
    * Insertion du couple (< K, P > dans temp *
    * Si n est une feuille, P pointe vers un bloc de données *
    * Si n est un nœud interne, P pointe vers un nœud fils *
    Insérer < K, P > dans temp à la bonne place
    Répartir le contenu de temp entre n et f
    Fin_4

* Si tous les frères de n sont pleins,*
* on crée un nouveau nœud new *
Sinon
        Début_5
        Soit new ← nil  * Le nouveau nœud *
        * new est insérée entre n et la feuille suivante de n*
        new.P_next ← n.P_next
        n.P_next ← new
        * Les clés de n sont réparties entre n et new *
        Copier n dans temp
        Insérer < K, Pr > dans temp à la bonne place
        Répartir le contenu de temp entre n et new
        K ← new.K_1
        * Si n est racine, Alors on crée une *
        * nouvelle racine dont n et new sont fils *
        Si S est vide Alors
            Début
            Soit root ←< n, K, new >
            fin_algo ← vrai
            Fin
        * Si n n'est pas racine, *
        * Alors on ajoute new dans l'arbre *
        Sinon
                Début_6
                * n reçoit le nœud père de n *
                n ← Sommet(S)
                Si n est non plein Alors
                    Début
                    Insérer (K, new) dans n à la bonne place
                    fin_algo ← vrai
                    Fin
                Sinon
                        Début
                        * le nœud père de new est placé *
                        * au sommet de S *
                        Ajouter n à S
                        * l'algorithme reprend (boucle tant que) *
```

```
                                                    * avec le couple (K, new) *
                                                    P ← new
                                                    Fin
                                                Fin6
                                            Fin5
                                        Fin3
                                    Fin2
                            Fin1
FIN
```

2 Algorithmes de recherche et d'insertion dans un arbre R

2.1 Recherche, dans un arbre R, des chemins menant au point P de l'espace

Cet algorithme est une adaptation de l'algorithme de recherche présenté dans [SVP+96].

```
DEBUT
    Soit C une pile initialisée à vide  * C va contenir les chemins menant au point P *
    Soit S une pile initialisée à vide
    Soit n un nœud initialisé à vide  * n est le nœud courant *
    Soit fin_algo un booléen initialisé à faux

    S ← la racine de l'arbre R
    Tant que fin_algo différent de vrai
            Début
            n ← Sommet(S)
            Si n est une feuille
               Début
               Alors Pour tous les couples (rem, oid) dans n Faire
                                          Si P appartient à rem Alors C ← C ∪ {oid}
               fin_algo ← vrai
               Fin
            * Si n n'est pas un nœud feuille on rappelle l'algorithme *
            Sinon
                  Pour tous les couples (rem, oid) dans n Faire
                                          Si P appartient à rem Alors S ← S ∪ {n.oid}
            Fin
Retourner C
FIN
```

2.2 Insertion d'un objet (rem, oid) dans un arbre R

Cet algorithme est une adaptation de l'algorithme de recherche présenté dans [SVP+96]. Cet algorithme permet d'insérer un objet de rectangle englobant minimum rem et d'adresse oid dans un arbre R.

```
DEBUT
    Soit S une pile initialisée à vide
    Soit n un nœud initialisé à vide  * n est le nœud courant *
    Soit fin_algo un booléen initialisé à faux
    Soit obj un objet spatial initialisé à vide  * obj est l'objet à insérer, il est de la forme (rem, oid) *
    Soit O ← nil  * O est une pile *
```

```
Ajouter (rem, oid) dans O  * Ajout dans la pile O du couple à insérer *
* Recherche(obj) est un appel à l'algorithme de recherche qui retourne les chemins menant aux feuilles *
* où insérer obj sous forme de pile *
Ajouter Sommet(Recherche(obj)) dans S  * S reçoit une feuille où insérer l'objet obj *

Tant que fin_algo différent de vrai
        Début1
        n ← Sommet(S)  * S est dépilée *
        obj ← Sommet(O)  * O est dépilée *
        Si n n'est pas plein Alors
            Début2
            Tant que n est non racine
                    Début3
                    Si obj ∉ n Alors Ajouter obj à n
                    * Si le rem de obj n'est pas inclus dans le rem de n *
                    * Alors il faut agrandir le rem de n *
                    Si obj.rem non inclus dans rem(n) Alors
                        Début4
                        rem(n) ← le rem contenant rem(n) et obj.rem
                        * Propagation de l'agrandissement au nœud père de n *
                        obj.rem ← rem(n)
                        obj.oid ← n
                        n ← Pere(n)
                        Fin4
                    Fin3
            fin_algo ← vrai
            Fin2
        * Si n est plein, Alors il faut créer un nouveau nœud *
        Sinon
            Début5
            Soit new ← nil  * Le nouveau nœud *
            Si n est racine Alors
                Début
                Soit racine ← nil  * La nouvelle racine de l'arbre*
                Répartir les entrées de n entre n et new
                rem(n) ← le rem de n après répartition
                rem(new) ← le rem de new après répartition
                Ajouter les couples (rem(n), n) et (rem(new), new) dans racine
                fin_algo ← vrai
                Fin

            * n n'est pas racine *
            Sinon
                    Début
                    * Le rem de n va être modifié, *
                    * il faut donc le supprimer de son nœud père *
                    * et insérer dans le nœud père n et new après répartition des entrées *
                    Supprimer (rem(n), n) de Pere(n)
                    Répartir les entrées de n entre n et new
                    rem(n) ← le rem de n après répartition
                    rem(new) ← le rem de new après répartition
```

```
                                    * L'algorithme est repris en insérant les couples (rem(n), n) et (rem(new), new) *
                                    * dans le nœud père de n
                                    Ajouter les couples (rem(n), n) et (rem(new), new) dans O
                                    Ajouter Pere(n) à S
                                    Fin
                            Fin5
                    Fin1
FIN
```