Entwurf und Implementierung mehrdimensionaler Zugriffsstrukturen

Vom Fachbereich Mathematik und Informatik
der Universität Bremen
zur Verleihung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte Dissertation

von

Bernhard Seeger

Bremen 1989

Danksagung

Viele Leute haben mir während der letzten Jahre, in denen ich die Dissertation geschrieben habe, geholfen. Besonders möchte ich meinem Betreuer Herrn Prof. Dr. Hans-Peter Kriegel meinen Dank für die ausgezeichnete fachliche und menschliche Betreuung aussprechen.

Bei meinem Zweitbetreuer, Herrn Prof. Dr. Peter Widmayer, möchte ich mich für die sehr hilfreichen Kommentare und Anmerkungen zu dieser Arbeit bedanken.

Ferner möchte ich mich bei meinen Kollegen Peter Heep, Stephan Heep, Michael Schiwietz und Ralf Schneider für ihre konstruktive Kritik an meiner Arbeit bedanken.

Weiterhin bin ich Herrn Prof. Dr. Peter C. Lockemann und Herrn Dr. Alfons Kemper für die sehr lehrreiche Zeit an der Universität Karlsruhe zu Dank verpflichtet.

Gutachter:

Prof. Dr. Hans-Peter Kriegel (Universität Bremen)
Prof. Dr. Peter Widmayer (Universität Freiburg)

Kolloquium: 20. Juni 1989

Inhaltsverzeichnis

1 Einführung

In den letzten Jahren sind Datenbanksysteme (DBS) primär für technisch wissenschaftliche Anwendungen entwickelt worden, wie z. B. in den Bereichen computerunterstütztes Konstruieren (CAD) [SRG 83], Robotik, Kartographie und Geologie [SW 86]. Früher entwickelte DBS, wie das einfache relationale DBS, wurden speziell für betriebswirtschaftliche Anwendungen konzipiert. Für oben genannte Anwendungen fehlt es diesen DBS insbesondere an adäquaten Datenmodellen, Benutzerfreundlichkeit und Effizienz.

Die Organisation der internen Ebene eines DBS bestimmt entscheidend seine Leistungsfähigkeit. Da Zugriffsstrukturen der wesentliche Bestandteil der internen Ebene sind, ist die Wahl geeigneter Zugriffsstrukturen für die Effizienz eines Datenbankentwurfs von zentraler Bedeutung. Eindimensionale Zugriffsstrukturen, wie z. B. B-Bäume [BM 72], unterstützen Suchoperationen, bei denen genau ein Attribut der Datensätze spezifiziert ist. Dagegen benötigen Anwendungen, in denen die Suche bzgl. mehrerer Attribute der Datensätze unterstützt werden soll, mehrdimensionale Zugriffsstrukturen. Insbesondere für geometrische Objekte ist dabei gefordert, daß nah beieinander liegende Datensätze auch physisch nah beieinander abgespeichert werden. Nur dadurch sind Raumanfragen, wie z. B. die Suche des nächsten Nachbars, mit möglichst wenig Leseoperationen auf dem Sekundärspeicher durchzuführen und somit effizient zu beantworten. Die invertierte Listenorganisation, die auch heute noch als mehrdimensionale Zugriffsstruktur in Datenbanksystemen verwendet wird, erfüllt diese Anforderung nur unzureichend [Kri 84]. Im folgenden werden wir Zugriffsstrukturen, die Datensätze bzgl. atomarer Attribute organisieren, als Punktzugriffsstrukturen (PZS) bezeichnen. Wie in der Literatur mehrfach aufgezeigt, so z. B. in [LS 87], hängt die Effizienz bisheriger mehrdimensionaler PZS, die die räumliche Lage der Datensätze berücksichtigen, wesentlich von der Verteilung und teilweise von der Eingabereihenfolge der Datensätze ab.

Neben mehrdimensionalen PZS werden in technisch wissenschaftlichen Anwendungen Zugriffsstrukturen benötigt, die komplexe Objekte verwalten können. Ein Beispiel für solche Objekte sind Parzellengrenzen, wie sie in kartographischen Anwendungen vorkommen. Da selbst mehrdimensionale PZS sich für die Organisation solcher Objekte nicht eignen, sind in letzter Zeit vermehrt spezielle Zugriffsstrukturen vorgestellt worden ([MT 83], [Gut 84], [NH 85], [SRF 87] [Gün 88], [SW 88], [SK 88]). Die Effizienz dieser Strukturen hängt primär von der Verteilung aber auch von der Komplexität der Objekte ab.

In dieser Arbeit werden neue PZS und Zugriffsstrukturen für komplexe Objekte vorgestellt, deren Leistungsverhalten nur unwesentlich von der Verteilung der Objekte beeinflußt wird. Die von uns entwickelten mehrdimensionalen dynamischen PZS - PLOP-Hashing und der Buddy-Hashbaum - werden wir detailliert betrachten. Dabei ist die Leistung beider Verfahren im wesentlichen unabhängig von der Reihenfolge der eingegebenen Datensätze und der Buddy-Hashbaum besitzt darüberhinaus ein von der Verteilung der Daten nahezu unabhängiges Verhalten.

Im zweiten Abschnitt werden wir zunächst näher auf die Struktur von PLOP-Hashing eingehen. PLOP-Hashing ist ein mehrdimensionales dynamisches Hashverfahren, das auf der Idee des linearen Hashings [Lit 80] basiert. Entsprechend linearem Hashing werden Datensätze ohne eine extern abgelegte Adreßtabelle organisiert. Wie wir noch zeigen werden, ist PLOP-Hashing sehr effizient für Datensätze mit nicht korrelierten Attributen, gleich welcher Verteilung die einzelnen Attribute folgen. Mit zunehmender Korrelation der Attribute degeneriert das Leistungsverhalten von PLOP-Hashing.

Das Konzept des Buddy-Hashbaums stellen wir im dritten Abschnitt vor. Die beim Buddy-Hashbaum verwendete Adreßtabelle ist als mehrdimensionaler Baum organisiert, wobei die internen Knoten des Baums ähnlich wie bei der Gitterdatei [NHS 84] verwaltet werden. Da das Einfügen, Löschen und die exakte Suche eines Datensatzes auf einen Pfad des Baums beschränkt bleibt, hängt die Leistung des Verfahrens von der Höhe des Baums ab. Wir zeigen auf, daß die Höhe des Buddy-Hashbaums dabei durch eine Funktion beschränkt ist, die von der Auflösung des Datenraums linear abhängig ist. Dieses Resultat gilt für beliebige Verteilungen.

Im vierten Abschnitt präsentieren wir die Resultate verschiedener Experimente, bei denen die Implementierungen mehrerer PZS getestet wurden. Dabei zeigt sich die Überlegenheit des Buddy-Hashbaums insbesondere bei extrem nicht-gleichverteilten Daten gegenüber der 2-Level-Gitterdatei [Hin 85] und PLOP-Hashing.

Im fünften Abschnitt gehen wir näher auf Zugriffsstrukturen für ausgedehnte Raumobjekte (Raumzugriffsstrukturen) ein. Zunächst nehmen wir eine Klassifikation existierender Raumzugriffsstrukturen vor. Desweiteren beschreiben wir die drei Techniken Clipping, überlappende Regionen und Transformation, mit denen eine beliebige mehrdimensionale PZS zu einer Raumzugriffsstruktur erweitert werden kann. Um die Vorteile der verschiedenen Techniken zu nutzen, schlagen wir eine auf PLOP-Hashing basierende Hybridmethode vor. Je nach Art der Raumobjekte werden die einzelnen Techniken in unserer Hybridmethode gewichtet, so daß Anfragen möglichst effizient beantwortet werden. Am Ende des Abschnitts präsentieren wir einen Leistungsvergleich verschiedener Raumzugriffsstrukturen.

Die wichtigsten Ergebnisse unserer Arbeit fassen wir im sechsten Abschnitt kurz zusammen und geben einen Ausblick auf unsere weiteren Ziele.

1.1 Globale Bezeichnungen und Definitionen

Eine Datei F entspricht einer Menge von Datensätzen $R^i = (K^i, I^i)$, $1 \leq i \leq n$, wobei n die Anzahl der Datensätze ist. Der d-dimensionale Schlüssel $K^i = (K_1^i, \ldots, K_d^i)$, $d \geq 1$, identifiziert den zugehörigen Datensatz R^i in der Datei F. Jede Schlüsselkomponente K_j^i ist einem linear geordneten und endlichem Wertebereich D_j entnommen, $1 \leq j \leq d$. Der Einfachheit halber nehmen wir an, daß D_j durch das halboffene Intervall $[min_j, max_j)$ gegeben ist. Als Datenraum D bezeichnen wir das kartesische Produkt der Wertebereiche D_j, $1 \leq j \leq d$. Insbesondere entspricht also der Datenraum einem d-dimensionalen Rechteck. Wir bezeichnen auch die Wertebereiche D_j, $1 \leq j \leq d$, als Achsen des Datenraums D.

Bei einem Datensatz R^i vernachlässigen wir den Informationsteil I^i, da dieser unter der Annahme, daß die Länge des Informationsteils konstant für alle Datensätze ist, keine Auswirkung auf die Position hat, wo der Datensatz in der Datei abgelegt wird. Bei variabel langen Datensätzen werden i. a. lokale Adreßtabellen benutzt, um die Position des Datensatzes zu bestimmen, wobei die relative Position in der Tabelle nur durch den Schlüssel bestimmt wird. Im folgenden wird vorausgesetzt, daß alle Datensätze in einer Datei gleich lang sind. Einen Datensatz können wir somit als einen d-dimensionalen

Punkt in dem Datenraum D veranschaulichen.

Eine Anfrage entspricht einem Prädikat, wobei alle Datensätze in der Datei F gesucht werden, die dieses Prädikat erfüllen. Je nach Art des Prädikats unterscheiden wir folgende Anfragen:

1. exakte Anfragen (exact match queries)
   Sei $A = (A_1, .., A_d) \in D$, finde alle Datensätze $R = (K_1, .., K_d, I)$ in der Datei F mit $K_j = A_j$, $1 \leq j \leq d$

2. partielle Anfragen (partial match queries)
   Sei $(i_1, .., i_d)$ eine Permutation von $(1, .., d)$, $k \in \{1, .., d\}$ und $A_{i_j} \in D_{i_j}$ für $1 \leq j \leq k$. Finde alle Datensätze $R = (K_1, .., K_d, I)$ in der Datei F mit $K_{i_j} = A_{i_j}$, $1 \leq j \leq k$

3. Bereichsanfragen (region queries)
   Sei $l_j, u_j \in D_j$ mit $l_j \leq u_j$, $1 \leq j \leq d$. Finde alle Datensätze $R = (K_1, .., K_d, I)$ in der Datei F mit $K_j \in [l_j, u_j]$, $1 \leq j \leq d$

4. partielle Bereichsanfragen (partial range queries)
   Sei $(i_1, .., i_d)$ eine Permutation von $(1, .., d)$, $k \in \{1, .., d\}$ und $l_{i_j}, u_{i_j} \in D_{i_j}$, $l_{i_j} \leq u_{i_j}$. Finde alle Datensätze $R = (K_1, .., K_d, I)$ in der Datei F mit $K_{i_j} \in [l_{i_j}, u_{i_j}]$, $1 \leq j \leq k$

Diese Anfragen können nun leicht geometrisch in dem rechteckigen Datenraum D veranschaulicht werden. So liegen z.B. alle Datensätze, die von einer Bereichsanfrage angesprochen werden, in einem rechteckigen Bereich des Datenraums D.

Zusätzlich zu diesen Anfragen betrachten wir folgende Operationen: das Löschen von Datensätzen aus einer Datei, das Einfügen von Datensätzen in eine Datei und der Verbund von mehreren Dateien zu einer Datei. Desweiteren betrachten wir ausschließlich dynamische Dateien, d. h. durch häufiges Löschen und Einfügen von Datensätzen verändert sich die Größe einer Datei ständig. Im Gegensatz dazu ist die Größe einer statischen Datei im wesentlichen konstant.

Im folgenden nehmen wir an, daß die Datensätze einer Datei im Sekundärspeicher auf Seiten verteilt werden, die exklusiv dieser Datei zugeordnet sind. Die Größe einer Seite ist dabei fest und beschränkt durch die Transfereinheit vom Haupt- in den Sekundärspeicher. Wir unterscheiden hierbei zwischen Daten- und Directoryseiten. In den Datenseiten einer Datei werden die Datensätze abgespeichert, während in den Directoryseiten Hilfsinformationen zur Unterstützung der Anfragen abgelegt werden. Als Seitenkapazität b, $b > 0$, bezeichnen wir die maximale Anzahl von Datensätzen, die in einer Datenseite abgespeichert werden können. In den meisten Systemen beträgt die Größe einer Seite zwischen 512 Bytes und 8K Bytes. In Abhängigkeit der Länge eines Datensatzes läßt sich daraus die Seitenkapazität berechnen. Die Größe einer Datei ist durch die Anzahl der belegten Seiten bestimmt.

In dieser Arbeit stellen wir Methoden vor, die Datensätze effizient im Sekundärspeicher organisieren. Dabei sollen einerseits die oben genannten Operationen und Anfragen schnell ausgeführt werden und andererseits eine hohe Speicherplatzausnutzung (Span) garantiert werden. Üblicherweise hängt die Ausführungszeit einer Operation im wesentlichen von der Zeit ab, um auf die benötigten Seiten im Sekundärspeicher zuzugreifen. Insbesondere ist also die Anzahl von Diskzugriffen auf den Sekundärspeicher ein gutes Maß für die Effizienz. Die Speicherplatzausnutzung (Span) entspricht dem Verhältnis von dem minimalen Platzbedarf für die Datensätze zu dem tatsächlich reservierten Speicherplatz für die Datensätze. Man beachte, daß es leicht ist, Methoden mit hoher Span oder mit sehr geringen Kosten für eine exakte Anfrage zu entwerfen. So wird z.B. bei einer einfachen sequentiellen Dateiorganisation eine fast 100% Span garantiert, wobei aber eine exakte Anfrage im Durchschnitt $n/2b$ Zugriffe erfordert. Speziell für die Beantwortung von komplexen Anfragen, wie z.B. partiellen Bereichsanfragen, benötigen wir Methoden, die eine hohe Span und geringe (exakte) Zugriffskosten garantieren. Wir bezeichnen diese Methoden wegen der Veranschaulichung von Datensätzen durch Punkte im d-dimensionalen Datenraum D als Punktzugriffsstrukturen (PZS). Für $d > 1$ sprechen wir von mehrdimensionalen PZS, andernfalls von eindimensionalen PZS.

Unser Ziel ist es hier primär PZS vorzustellen, die (partielle) Bereichsanfragen unterstützen. Da nun nah beieinander liegende Datensätze sehr häufig gemeinsam das Prädikat einer Bereichsanfrage erfüllen, muß eine effiziente Punktzugriffsstruktur garantieren, daß diese Datensätze im Speicher physisch nah beieinander liegen. Wenn möglich, sollten diese Sätze gemeinsam in einer Datenseite liegen. Andernfalls sollten die entsprechenden Seiten in einfacher Weise miteinander verbunden sein.

Als Maß für die Effizienz einer komplexen Anfrage können wir nicht mehr einfach die Anzahl der Zugriffe nehmen, da diese von der Anzahl der Antworten abhängt. Deshalb verwenden wir als Indikator für die Effizienz von komplexen Anfragen den folgenden Parameter:

$$\frac{\text{Anzahl der Anworten}}{(\text{Anzahl der Diskzugriffe}) * b}$$

Wie oben bereits erwähnt betrachten wir ausschließlich dynamische Dateien. Deshalb fordern wir von einer PZS, daß ein Anwachsen bzw. Verkleinern einer Datei unterstützt wird, ohne dabei wesentlich an Effizienz zu verlieren. Eine Datei wird durch eine Reorganisation vergrößert bzw. verkleinert, wobei für die Zeitdauer einer Reorganisation keine weiteren Operationen auf der Datei erlaubt sind. Daher sollten (globale) Reorganisationen, die einen hohen Aufwand erfordern, vermieden werden. Globale Reorganisationen werden häufig bei sogenannten statischen Hashverfahren angewendet. Im Gegensatz dazu wird bei dynamischen PZS die Reorganisation schrittweise vorgenommen, wobei jede Teilreorganisation wenig Aufwand erfordert. Wichtige Techniken für dynamische PZS sind das Splitten und Verschmelzen von Seiten. Beim Splitten werden die Datensätze einer Seite in zwei Gruppen aufgeteilt, wobei die Gruppen jeweils separat voneinander in zwei Seiten abgelegt werden. Dabei sollte zum einen die Entscheidung in welcher Seite der Datensatz liegt nur vom Schlüssel abhängen, und zum anderen sollte jede Gruppe in etwa gleich viele Datensätze besitzen. Unter dem Verschmelzen von zwei Seiten verstehen wir gerade die inverse Operation des Splitten.

Bei den meisten PZS hängt die Leistung von der jeweiligen Verteilung der Schlüssel ab. Im folgenden sei nun Φ die Verteilungsfunktion der Schlüssel $(K_1, \ldots, K_d)$ in dem Datenraum D und ϕ_j die Verteilungsfunktion der Schlüsselkomponente K_j in dem Wertebereich D_j, $1 \leq j \leq d$. Wir bezeichnen die Verteilung eines mehrdimensionalen Schlüssels als unabhängig, falls

$$\Phi(K_1, \ldots, K_d) \;=\; \prod_{j=1}^{d} \phi_j(K_j) \qquad \forall\ (K_1, \ldots, K_d) \in D$$

gilt. Andernfalls sprechen wir von einer abhängigen Verteilung, bzw. von korrelierten Schlüsselkomponenten.

1.2 Ein Überblick von Punktzugriffsstrukturen

Da die Effizienz eines Datenbanksystems im wesentlichen von der Effizienz der vorhandenen Zugriffsstrukturen abhängt, ist die Entwicklung solcher Zugriffsstrukturen von zentraler Bedeutung. Die Effizienz einer Punktzugriffsstruktur (PZS) hängt natürlich von den jeweiligen Anforderungen ab, so daß es bei dem breiten Anwendungsspektrum eine entsprechend große Anzahl von PZS gibt. Beschränkt man sich nun auf PZS, die für die oben spezifizierten Anforderungen geeignet sind, so ist es selbst für einen Fachmann auf diesem Gebiet schwierig, einen genauen Überblick zu geben. Als Kriterium für die Zuordnung mehrdimensionaler PZS zu verschiedenen Klassen nehmen wir die Art, wie die PZS den Datenraum unterteilen.

PZS, die das Prinzip von binären Bäumen verallgemeinern, haben wir hierbei nicht berücksichtigt, da diese Strukturen zwar für die Organisation von Daten im Hauptspeicher sehr gut geeignet sind, für die Organisation von Daten im Sekundärspeicher aber nicht. Typische Vertreter dieser Gruppe von Verfahren sind der kd-Baum [Ben 75], [Ben 79], BD-Baum [OsS 83], der Quadrantenbaum (quad-tree) [FB 74, Sam 85] und der digitale kd-Baum (kd-trie) [Ore 82].

Ausschließlich alle mehrdimensionalen dynamischen PZS basieren auf eindimensionalen PZS, wie z. B. B-Bäumen oder eindimensionalen Hashverfahren. Das Prinzip von mehrdimensionalen PZS ist, den Datenraum D in m Regionen (Seitenregionen), $S_1, \ldots, S_m$, $m > 0$, aufzuteilen, so daß alle Datensätze einer Seitenregion S_i, $1 \leq i \leq m$, in einer Seite oder in einer kurzen Kette von Seiten gehalten werden können. Die Anzahl m der Seitenregionen hängt dabei linear von der Anzahl der Datensätze in der Datei ab. Unter einer Region verstehen wir hierbei eine endliche Vereinigung atomarer d-dim. konvexer geometrischer Objekte, die ganz im Datenraum D liegen. Bei den meisten PZS entsprechen diesen atomaren Objekten d-dim. Rechtecke. Im Gegensatz dazu verwendet z. B. der Zellbaum [Gün 88] konvexe Polygone, Tri-Zellverfahren [FR 87] d-dim. Dreiecke. Entsprechend den Kriterien, ob die Aufteilung in Seitenregionen disjunkt oder nicht disjunkt ist, ob die Region aus mehreren atomaren Objekten oder genau einem Objekt besteht, oder ob die Aufteilung vollständig (d. h. $\bigcup_{i=1}^{m} S_i = D$) oder nicht vollständig ist, erhalten wir fünf Klassen von PZS.

In der Tabelle 1 haben wir existierende PZS in die einzelnen Klassen (K1) - (K5) eingeordnet. Beispielhaft haben wir die Unterteilung des Datenraums einzelner Verfahren in entsprechenden Abbildungen (siehe Abb. 1) veranschaulicht. Wir gehen zunächst auf die mächtigste Klasse (K1) im folgenden näher ein.

Dynamische PZS benutzen im Prinzip zwei verschiedene Techniken, um die Position eines Datensatzes in der Datei zu bestimmen. Die eine Technik basiert auf Vergleich mit zusätzlich abgespeicherten Daten (Separatoren), während die andere Technik Positionen in der Datei durch Auswertung von Funktionen bestimmt. Beispiele hierfür sind die eindimensionalen PZS B^+-Baum [BM 72, Com 79] und lineares Hashing [Lit 80]. Entsprechend dieser Unterscheidung könnte man nun jede Klasse (K1) - (K5) nochmals in drei Gruppen von Verfahren aufteilen, nämlich Verfahren, die ausschließlich eine der obengenannten

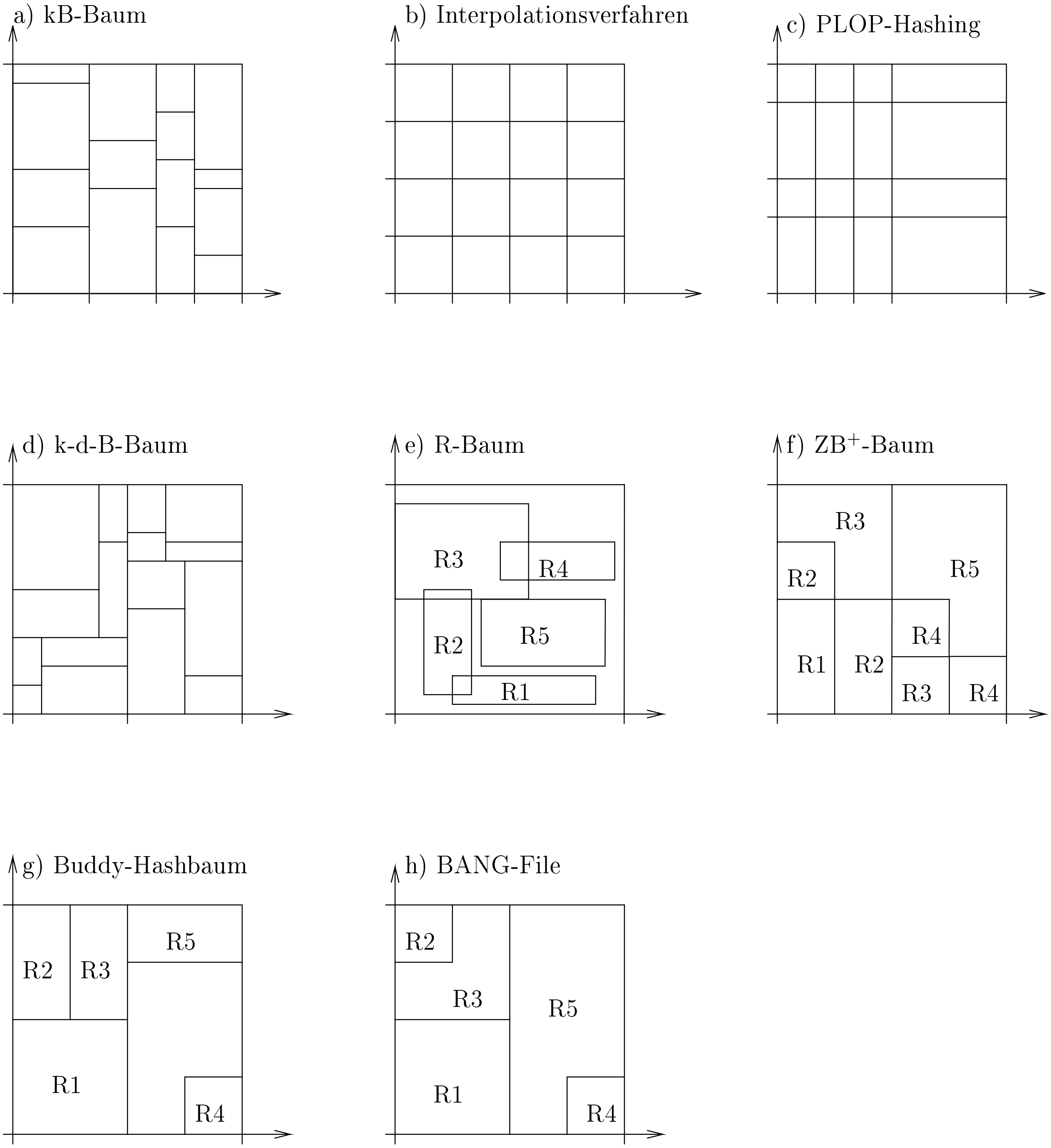
Techniken, bzw. beide Techniken gemeinsam verwenden. Die effizientesten PZS benutzen i. a. sowieso beide Techniken gemeinsam, so daß eine feinere Klassifizierung nicht sinnvoll ist.

Da der B-Baum [BM 72] sich sehr gut als eindimensionale PZS bewährte, wurde zunächst versucht diese Struktur mit ihren Eigenschaften für mehrdimensionale Verfahren zu verallgemeinern. Der B-Baum ist eine PZS, die ausschließlich durch Vergleich mit Separatoren Datensätze organisiert. Die ersten vielversprechenden Verfahren wie mehrdimensionale B-Bäume [SO 82] oder kB-Bäume [Kri 84] haben den Nachteil die Daten lexikographisch zu organisieren, d. h. ähnlich zu den invertierten Listen ist bei diesen Verfahren eine Schlüsselkomponente dominant zu den anderen Komponenten. Daher können partielle Anfragen, bei denen die dominante Achse nicht spezifiziert ist, nur sehr ineffizient beantwortet werden. Diese Verfahren sind die bisher einzigen mehrdimensionalen PZS, die die Eigenschaften des B-Baums für mehrdimensionale Daten verallgemeinern. Die exakte Suche, das Löschen und Einfügen von Datensätzen benötigen O(log n) Diskzugriffe, wobei gleichzeitig die Höhe des Baums in O(log n) anwächst. Auf Grund der lexikographischen Sortierung sind diese Verfahren i. a. nicht für eine effiziente Beantwortung von Bereichsanfragen geeignet. Der k-d-B-Baum [Rob 81] stellt eine mehrdimensionale Verallgemeinerung des B^+-Baums dar, besitzt aber nicht die Eigenschaften des B^+-Baums. So kann der k-d-B-Baum nicht eine Span über 50% garantieren. Zudem erfordert die Aufrechterhaltung der Balancierung des k-d-B-Baums im schlechtesten Fall lineare Zeit. Eine interessante Verallgemeinerung des k-d-B-Baums ist der hB-Baum [LS 87]. Unter der Voraussetzung, daß Datensätze nur in die Datei eingefügt und nicht mehr aus der Datei entfernt werden, garantiert der hB-Baum eine hohe Span sowie eine logarithmische Höhe des Baums.

Die ersten mehrdimensionalen dynamischen Hashverfahren (MDH), wie mehrdimensionales erweiterbares Hashing [Tam 82, Oto 84] und das Interpolationsverfahren [Bur 83, OS 83], basieren ausschließlich auf dem Prinzip der Adreßberechnung. Mit Hilfe einer Hashfunktion wird dabei - ähnlich wie bei linearem Hashing [Lit 80], bzw. erweiterbarem

Klasse	*Eigenschaften*			*Verfahren*
	atomar	vollständig	disjunkt	
(K1)	×	×	×	Gitterdatei, Interpolationsverfahren, interpolierte Gitterdatei, k-d-B-Baum, kB-Baum, mehrdimensionaler B-Baum, erweiterbarer Hashbaum, mehrdimensionales erweiterbares Hashing, MOLHPE, Quantil-Hashing, PLOP-Hashing
(K2)	×			R-Baum
(K3)	×		×	Buddy-Hashbaum, R^+-Baum
(K4)	×	×		Zwillings-Gitterdatei
(K5)		×	×	BANG-File, Z-Ordnungs-Verfahren, hB-Baum

Tabelle 1: Klassifizierung von mehrdimensionalen PZS in fünf Klassen



a) kB-Baum

b) Interpolationsverfahren

c) PLOP-Hashing

d) k-d-B-Baum

e) R-Baum

f) ZB$^+$-Baum

g) Buddy-Hashbaum

h) BANG-File

Abbildung 1: Partitionierung des 2-dimensionalen Datenraums durch verschiedene mehrdimensionale PZS

Hashing [FNPS 79] - eine Adresse direkt in einer Datendatei, bzw. in einer Adreßtabelle berechnet. Eine Adreßtabelle bezeichnen wir im folgenden auch als Directory. Das Problem hierbei ist, daß im Prinzip der Datenraum durch ein orthogonales Gitter in disjunkte Regionen gleicher Größe aufgeteilt wird, so daß für gleichverteilte Schlüssel diese Verfahren sehr effizient sind, während für nicht-gleichverteilte Schlüssel die Leistung degeneriert. Weitere effizientere Varianten dieser Gruppe von Verfahren sind mehrdimensionales lineares Hashing mit partiellen Erweiterungen [KS 86] und das global ordnungserhaltende mehrdimensionale lineare Hashing [HSW 88a].

Es ist schnell erkannt worden, daß mehrdimensionale PZS insbesondere bei gleichzeitiger Verwendung von Adreßberechnung und Separatoren effizient sein können. Bei diesen Hybridstrukturen unterscheiden wir nochmals zwei Typen von Verfahren: Gitterverfahren und Hashbäume.

Die Gitterverfahren partitionieren zwar wie die oben erwähnten MDH den Datenraum mit Hilfe eines Gitters, das Gitter selbst paßt sich aber an die Verteilung der Daten an. Die explizite Beschreibung des Gitters wird nun in Listen, bzw. Bäumen resident im Hauptspeicher gehalten. Bei Suchoperationen müssen nun zunächst diese Listen durchsucht werden, d. h. es werden Schlüsselkomponenten mit Separatoren verglichen, und als Ergebnis bekommt man eine partielle Adresse (Index) für jede Achse geliefert. Durch Eingabe dieser Indizes in eine Hashfunktion erhält man schließlich die gewünschte Adresse einer Datenseite in einer Datei oder einer Komponente in einer Adreßtabelle. Sowohl die Gitterdatei [NHS 84], eine der bekanntesten mehrdimensionalen dynamischen PZS, als auch das Quantil-Hashing [KS 87] und PLOP-Hashing [KS 88] sind typische Vertreter dieser Gruppe von Verfahren. In diesem Abschnitt beschränken wir unsere Betrachtungen auf die Gitterdatei und PLOP-Hashing. Die Gitterdatei basiert auf dem Prinzip von erweiterbarem Hashing, während Quantil- und PLOP-Hashing das Prinzip von linearem Hashing für mehrdimensionale Daten verallgemeinern. Der Unterschied von PLOP-Hashing zu der Gitterdatei entspricht deshalb in etwa dem Unterschied von linearem Hashing zu erweiterbarem Hashing. PLOP-Hashing, bzw. lineares Hashing besitzen im Gegensatz zu der Gitterdatei, bzw. erweiterbarem Hashing keine extern abgelegten Hilfsinformationen. In diesem Zusammenhang wird auch von PZS ohne Directory (PLOP-Hashing) und PZS mit Directory (Gitterdatei) gesprochen. In den folgenden Abschnitten werden wir sehr ausführlich auf die Struktur von PLOP-Hashing eingehen, insbesondere die wichtigsten Algorithmen angeben und mögliche Varianten von PLOP-Hashing aufzeigen. Der Grund für uns, neben der Gitterdatei noch ein Verfahren dieser Gruppe vorzuschlagen, sind folgende Nachteile der Gitterdatei:

1. Das Directory der Gitterdatei wächst bereits für unabhängig gleichverteilte Schlüssel in $O(n^{1+(d-1)/(b*d)})$ [Reg 85], wobei n die Anzahl der Datensätze in der Datei, d die Dimension der Schlüssel und b die Seitenkapazität ist. Für nicht-gleichverteilte Daten kann das Directory zumindest in $O(n^d)$ anwachsen.

2. Die dynamische Erweiterung des Directory kostet $O(N^{1-1/d})$ Diskzugriffe, wobei N der Größe des Directory entspricht. Diese Erweiterungen müssen in einem Schritt ausgeführt werden, d. h. andere Operationen können während einer Erweiterung nicht ausgeführt werden.

3. Es wird zwar garantiert, daß eine exakte Anfrage in zwei Diskzugriffen beantwortet wird, aber diese zwei Zugriffe werden auch stets benötigt.

Zu bemerken ist, daß diese Nachteile der Gitterdatei bereits in ähnlicher Form bei erweiterbarem Hashing vorhanden sind. Da sowohl PLOP-Hashing als auch die Gitterdatei den Datenraum mit Hilfe eines orthogonalen Gitters partitionieren, werden diese Verfahren ihre optimale Leistung höchstens für unabhängig verteilte Schlüssel erbringen können. Für abhängige verteilte Schlüssel wird sich die Leistung erheblich verschlechtern.

Dieser Nachteil der Gitterverfahren war der Hauptgrund für die Entwicklung von Hashbäumen. Da das schnelle Anwachsen der Adreßtabelle der Hauptnachteil der Gitterdatei ist, wurde bereits bei den ersten Realisierungen [Hin 85] eine weitere Adreßtabelle eingeführt, die zur Organisation der ursprünglichen Adreßtabelle benutzt wurde. Diese 2-Level Architektur wurde auch bereits für mehrdimensionales erweiterbares Hashing [MT 83] vorgeschlagen. Verallgemeinert man dieses Prinzip, so erhält man ein Directory, einen sogenannten Hashbaum, der ähnlich zu einem k-d-B-Baum organisiert ist. Jeder Knoten des Hashbaums gehört zu einem rechteckigen Teil des Datenraums, wobei der zu einem inneren Knoten gehörende Teilraum mit Hilfe eines Gitters unterteilt wird. Insbesondere muß bei jedem Knoten durch eine Hashfunktion eine Adresse berechnet werden. Im Vergleich zu der Gitterdatei erfordert die Beantwortung einer exakten Anfrage i. a. mehr als zwei Zugriffe. Bekannte Verfahren dieser Gruppe sind der erweiterbare Hashbaum [Oto 86] oder die interpolierte Gitterdatei [Ouk 85]. Diese Verfahren unterscheiden sich im Prinzip nur in der Organisation der inneren Knoten des Hashbaums. Die Leistung bei stark korrelierten Daten ist zwar bei diesen Verfahren im Vergleich zu der Gitterdatei schon verbessert, trotzdem können z. B. beim erweiterbaren Hashbaum Einträge existieren, zu denen es keine Datensätze gibt.

In letzter Zeit ist erkannt worden, daß PZS der Klasse (K1) die Anforderung nach gleichbleibender Effizienz bei beliebiger Verteilung der Daten nicht erfüllen können. So werden wir in Abschnitt 3 eine PZS der Klasse (K3), den Buddy-Hashbaum, vorstellen, der den Datenraum durch Seitenregionen nicht ganz abdeckt, sondern gerade die Bereiche, in welchen keine Daten liegen, nicht berücksichtigt. Wie bei Verfahren der Klasse (K1), fordert der Buddy-Hashbaum auch rechteckige Seitenregionen. Die Multi-Level Gitterdatei [WK 85] besitzt ein zum Buddy-Hashbaum ähnliches Konzept. Einige Unterschiede in den Strukturen von Buddy-Hashbaum und Multi-Level Gitterdatei haben aber einen wesentlichen Einfluß auf die Effizienz der Verfahren. Im Gegensatz zum R^+-Baum [SRF 87] bietet der Buddy-Hashbaum ein verbessertes dynamisches Verhalten. Insbesondere das Löschen von Datensätzen und die damit verbundene Reorganisation der Daten werden beim Buddy-Hashbaum besser unterstützt. Der R^+-Baum ist zwar primär für die Organisation von mehrdimensionalen Rechtecken vorgeschlagen worden, er kann aber ebenfalls zur Organisation von mehrdimensionalen Punkten eingesetzt werden.

Entsprechend zum R^+-Baum ist der R-Baum [Gut 84] ein Verfahren zur Organisation von Rechtecken. Als PZS der Klasse (K2) bietet der R-Baum viele Freiheiten in der Partitionierung des Datenraums. Insbesondere beim Löschen von Datensätzen kann dies die Leistung negativ beeinflußen. Durch eine spezielle globale Reorganisation, die als Packen [RL 85] bezeichnet wird, kann die Effizienz eines statischen R-Baums erheblich verbessert werden.

Das Konzept der Zwillings-Gitterdatei [HSW 88b] ist die Organisation einer Datei mit zwei voneinander abhängigen Gitterdateien. Beim Einfügen eines Datensatzes wird versucht, den Datensatz bestmöglich in einer der beiden Gitterdateien unterzubringen. Der wesentliche Vorteil ist eine höhere Span und die damit verbundene Leistungsteigerung bei partiellen Bereichsanfragen. Dieses Konzept ist nicht nur für die Gitterdatei, sondern für beliebige PZS verwendbar.

Interessante Ansätze für mehrdimensionale PZS weisen die Verfahren der Klassen (K5) auf. Ziel dieser Verfahren, wie BANG-File (balanced and nested grid file) [Fre 87, Fre 88], ZB^{+}-Baum [OM 84] oder hB-Baum (holey brick tree) [LS 87], ist es, eine flexiblere Form von Seitenbereichen zu erlauben. Dadurch wird eine hohe Span sowie ein linear anwachsendes Directory garantiert. Der hB-Baum organisiert seine Daten mit Hilfe eines kd-Baums [Ben 79], der in einem k-d-B-Baum [Rob 81] eingebettet ist. Einige Probleme des k-d-B-Baums besitzt auch der hB-Baum, z. B. kann das Löschen von Datensätzen erheblich die Leistung mindern. Sowohl das BANG-File als auch der ZB^{+}-Baum organisieren die inneren Knoten mit Hilfe der Z-Ordnung [OM 84]. Nachteil von all diesen Verfahren ist, daß Seitenregionen nicht notwendigerweise zusammenhängend sind. Dadurch werden Datensätze, die weit auseinander liegen, teilweise in einer Datenseite gehalten. Wir erwarten also insbesondere bei komplexen Anfragen, daß relativ viele Datensätze überprüft werden müssen.

1.3 Lineares Hashing

Mehrdimensionale dynamische Hashverfahren ohne Directory können als mehrdimensionale Verallgemeinerung des linearen Hashings (LH) [Lit 80] verstanden werden. Um die grundsätzlichen Ideen von dynamischen Hashverfahren ohne Directory in einfacher Weise aufzuzeigen, stellen wir deshalb zunächst kurz die Konzepte von linearem Hashing vor.

Bei LH wird mit Hilfe einer Hashfunktion $H_m : D \longmapsto \{0, \ldots, m-1\}$ eine Adresse einer Kette von Seiten berechnet, wobei die Anzahl m der Ketten, $m \geq 1$, von der Anzahl n der Datensätze in der Datei abhängt. Da kein Directory benutzt wird, muß LH sogenannte Überlaufsätze zulassen. Dies sind Datensätze, die in der ersten Seite der Kette, die auch als Primärseite bezeichnet wird, keinen Platz finden und deshalb in einer anderen Seite der Kette (Sekundärseite) abgelegt werden. Primär- und Sekundärseiten werden in zwei separaten Dateien organisiert, die wir entsprechend als Primär- und Sekundärdatei bezeichnen. Diese einfache Methode Überlaufsätze zu verwalten, wird als Seitenverkettung bezeichnet.

Für das Erweitern bzw. Verkleinern der Datei bedient sich LH der Technik des Splittens bzw. Verschmelzens von Seiten und Ketten. Der Zeitpunkt, wann ein Splitten oder Verschmelzen stattfindet, ist durch eine Regel festgelegt, die wir als Kontrollfunktion bezeichnen. So kann z. B. die Datei immer dann um eine Kette erweitert, bzw. verkleinert werden, falls die Span eine feste obere Schranke übersteigt, bzw. unter eine feste untere Schranke absinkt. Als Folge dessen muß nach jeder Einfüge- und Löschoperation die Kontrollfunktionen angewendet werden, so daß die Span stets zwischen diesen beiden Schranken liegt. Ein Expansionszeiger ep verweißt stets auf die Kette (genauer gesagt auf die Primärseite der Kette), die, falls von der Kontrollfunktion ein Erweitern der Datei verlangt wird, als nächstes gesplittet werden soll. Nehmen wir an, daß bei Initialisierung der Datei genau eine Kette (mit einer leeren Primärseite) existiert und deshalb ep (= 0) auf diese Kette verweist. Eine Datei wird linear erweitert, falls nach jeder Erweiterung der Datei, ep wie folgt neu bestimmt wird:

1. Falls die Dateigröße sich verdoppelt hat, d. h. die Datei besteht aus 2^L Ketten mit Adressen $0, \ldots, 2^L - 1$, $L > 0$, wird eine Folge $\{p_j\}_{j=0}^{2^L-1}$ mit $0 \leq p_j < 2^L$ und $p_i \neq p_j$ für $i \neq j$ bestimmt. Danach wird ep auf p_0 gesetzt.
2. Andernfalls wird ep auf p_{j+1} gesetzt, $0 \leq j \leq 2^L - 2$.

Der Parameter L, der als Level der Datei bezeichnet wird, zeigt hierbei an, wie oft sich die Dateigröße verdoppelt hat. Für LH [Lit 80] wurde vorgeschlagen $p_j = j$, $0 \leq j < 2^L$, zu wählen. Sei nun eine Datei, die durch LH organisiert wird, mit genau 2^L Ketten ($ep = 0$) gegeben. Falls eine Erweiterung durch die Kontrollfunktion gefordert wird, so wird die Kette mit Adresse 0 in die Ketten mit Adressen 0 und 2^L aufgesplittet und ep wird inkrementiert ($ep = 1$). Bei der nächsten Erweiterung der Datei wird die Kette 1 in die Ketten 1 und $2^L + 1$ aufgesplittet, etc. .

Beim Splitten einer Kette wird versucht, die Hälfte der Datensätze in eine neu bereitgestellte Kette umzuspeichern. Dadurch ergibt sich, daß die Belegung von bereits geplitteten Ketten, dies sind die Ketten mit Adressen $0, \ldots, ep-1$ und $2^L, 2^L+1, \ldots, 2^L+ep-1$, im Durchschnitt nur halb so hoch ist wie der Belegungsfaktor einer anderen Kette. Um eine gleichmäßigere Belegung der Ketten zu gewährleisten, wurde in [Lar 80] das Konzept partieller Erweiterungen für LH vorgeschlagen. In einem späteren Abschnitt werden wir partielle Erweiterungen auf ein mehrdimensionales dynamisches Hashverfahren ohne Directory übertragen.

Die Wahl der Hashfunktionen H_i, $1 \leq i \leq m$, wirkt sich auf die Eigenschaften des LH aus. Wir stellen hier eine ordnungsbewahrende Hashfunktion vor. Bei Benutzung einer ordnungsbewahrenden Hashfunktion können die Ketten p_i, $0 \leq i < m$, so sortiert werden, daß alle Schlüssel der Kette $p_{j_{i-1}}$ kleiner sind als die Schlüssel der Kette p_{j_i}, $1 \leq i < m$, $j_i \neq j_k$ für $i \neq k, 0 \leq i, k < m$. Diese Variante von LH wurde in [Ore 83] vorgestellt. Da eine unserer wichtigsten Forderungen die Unterstützung von Bereichsanfragen ist, kommt der Ordnungsbewahrung wesentliche Bedeutung zu. Sei das Einheitsintervall [0,1) der Datenraum und $\sum_{j\geq 1} b_j 2^{-j}$ die binäre Repräsentation des Schlüssels K, so ist eine ordnungsbewahrende Hashfunktion für LH wie folgt gegeben:

$$H_m(K) = \begin{cases} \sum_{j=0}^{L-1} b_{j+1} 2^j & falls \ \sum_{j=0}^{L-1} b_{j+1} 2^j < 2^{L-1} + ep \\ \\ \sum_{j=0}^{L-2} b_{j+1} 2^j & sonst \end{cases}$$

Zu beachten ist hierbei, daß das Level L = $\lfloor \log_2 m \rfloor$ und der Wert des Expansionszeigers $ep = m - 2^L$ durch m bestimmt sind. Das Problem für ordnungsbewahrendes LH ist, daß der Wertebereich dynamisch in gleichgroße Intervalle aufgeteilt wird, so daß für nicht gleichverteilte Schlüssel die Leistung degeneriert. Diese Schwierigkeit ergibt sich i. a. bei dem üblichen LH nicht, da durch geeignete Transformationen der Daten Nichtgleichverteilungen z. T. beseitigt werden können. Diese Transformationen besitzen aber nicht mehr die ordnungsbewahrende Eigenschaft.

2 PLOP-Hashing

2.1 Die statische Datenstruktur

Um eine einfache Einführung in die Struktur des PLOP-Hashing zu geben, setzen wir in diesem Abschnitt zunächst einen statischen Zustand der Datei voraus, d. h. die Anzahl n der Datensätze, $n > 0$, als auch die Anzahl m der Ketten, $m > 0$, ist konstant. Auch wollen wir, wie für LH beschrieben, Überlaufsätze durch Seitenverkettung organisieren.

PLOP-Hashing unterteilt den d-dimensionalen Datenraum $D = \times_{j=1}^{d} [min_j, max_j)$ mit Hilfe eines orthogonalen Gitters in disjunkte und rechteckige Zellen $C_0, \ldots, C_{m-1}$. Alle Datensätze, die in einer Zelle liegen, werden in einer Kette von Seiten abgelegt. Andererseits enthält solch eine Kette nur Datensätze aus genau einer Zelle. Ein orthogonales Gitter besteht aus (d-1)-dimensionalen (orthogonalen) Hyperebenen, die sich wie folgt beschreiben lassen:

$$H_j(A) := \{x \in D \mid x_j = A\} \qquad\qquad A \in [min_j, max_j),\ 1 \leq j \leq d$$

Eine orthogonale Hyperebene zerschneidet also den Datenraum in zwei rechteckige Teilräume. Den eindimensionalen Wert $A \in [min_j, max_j)$ einer Hyperebene $H_j(A)$ bezeichnen wir als Partitionierungspunkt der j-ten Achse, $1 \leq j \leq d$. Weiterhin sei P_j die Menge aller Partitionierungspunkte der j-ten Achse und $\overline{P}_j = P_j \cup \{min_j, max_j\}$, $1 \leq j \leq d$. Ein orthogonales Gitter G_D ist dann wie folgt gegeben:

$$G_D := G_D(\overline{P}_1, \ldots, \overline{P}_d) := \{x \in D \mid \exists j \in \{1, \ldots, d\} : x_j \in \overline{P}_j\}$$

Sei $m_j =\mid P_j \mid$ die Anzahl der Partitionierungspunkte der j-ten Achse, so ist die Anzahl m von Zellen des Gitters G_D durch

$$m = \prod_{j=1}^{d} (m_j + 1)$$

gegeben. Seien $A, B \in \overline{P}_j$, $A < B$, zwei benachbarte Partitionierungspunkte (d. h. es gibt kein $C \in \overline{P}_j$ mit $A < C < B$), so verstehen wir unter einer Scheibe der j-ten Achse den rechteckigen Teilraum $\{x \in D \mid A \leq x_j < B\}$, der durch die zwei Hyperebenen $H_j(A)$ und $H_j(B)$, $1 \leq j \leq d$, begrenzt ist. Zu einem Gitter $G_D(\overline{P}_1, \ldots, \overline{P}_d)$ gibt es in der j-ten Achse $m_j + 1$ Scheiben, $1 \leq j \leq d$.

Betrachten wir die Situation in Abb. 2, in der eine Datei mit 26 2-dimensionalen Datensätze dargestellt ist. Die Datensätze sind dabei durch Punkte veranschaulicht. Der Datenraum selbst ist in 16 Zellen aufgeteilt. Für eine Seitenkapazität von 2 Datensätzen (b=2), erhalten wir insgesamt zwei Überlaufsätze, die in der linken oberen und rechten unteren Zelle vorzufinden sind. In jeder Achse hat PLOP-Hashing 4 Scheiben und dementsprechend drei Partitionierungspunkte erzeugt. Wir erhalten somit $P_1 = \{.2, .4, .8\}$ und $P_2 = \{15, 40, 60\}$.

Unsere erste Aufgabe ist es, eine geeignete Adreßfunktion $L : D \mapsto \{0, \ldots, m-1\}$ zu finden, so daß alle Datensätze einer Zelle genau einer Adresse $0, \ldots, m-1$ zugeordnet werden. Unter der Annahme einer statischen Datei (m ist fest), benötigen wir zunächst nur eine (statische) Adreßfunktion und nicht wie bei linearem Hashing eine

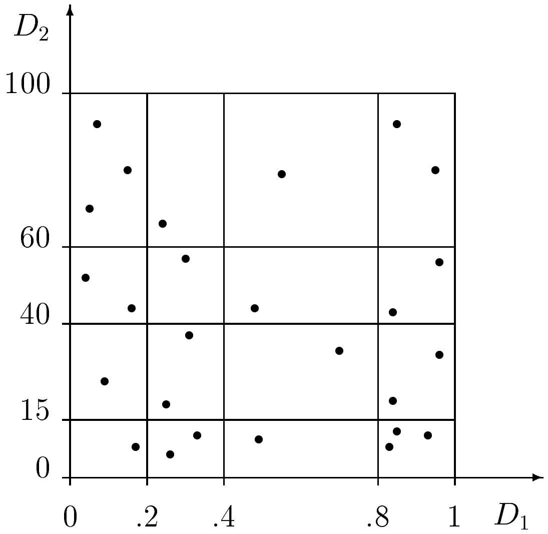


Abbildung 2: Partitionierung des 2-dimensionalen Datenraums [0.0,1.0)×[0,100) in 16 Zellen

von m abhängige Folge von Funktionen. Wir wollen hier die lexikographische Adreßfunktion vorstellen, die wir im folgenden Abschnitt für dynamische Dateien erweitern werden. Bei der lexikographischen Adreßfunktion benötigen wir zunächst eine 1:1 Abbildung zwischen den Scheiben der j-ten Achse und der ganzzahligen Menge $\{0, \ldots, m_j\}$, $1 \le j \le d$. Diese Abbildung muß explizit in sogenannten Skalen gehalten werden. Für $\{index_0, \ldots, index_{m_j}\} = \{0, \ldots, m_j\}$ sind die Skalen von der Form

$$(min_j, index_0, A_1, index_1, A_2, \ldots, A_{m_j}, index_{m_j}, max_j)$$

wobei $A_i \in P_j$, $1 \le i \le m_j$, ein Partitionierungspunkt der j-ten Achse ist und $A_i < A_{i+1}$ für $i \in \{1, \ldots, m_j - 1\}$, $1 \le j \le d$. Da wir nun $index_i$ der Scheibe $\{x \in D \mid A_i \le x_j < A_{i+1}\}$ zuordnen können, verwenden wir für Scheiben die Notation $S(index_i, j)$, $0 \le i < m_j$, $1 \le j \le d$. Dabei bezeichnen wir $index_i$ auch als partielle Adresse oder Index. Für unser Beispiel aus Abb. 2 nehmen wir an, daß die Skalen folgendes Aussehen haben:

$$\begin{array}{rlrlrlrlrlr} 1.Achse: & (& 0.0, & \mathbf{0}, & 0.2, & \mathbf{3}, & 0.4, & \mathbf{1}, & 0.8, & \mathbf{2}, & 1.0) \\ 2.Achse: & (& 0, & \mathbf{3}, & 15, & \mathbf{0}, & 40, & \mathbf{1}, & 60, & \mathbf{2}, & 100) \end{array} \tag{1}$$

Zur Verdeutlichung haben wir die Indizes der Achsen hervorgehoben.

Die Adresse für einen Schlüssel $K \in D$ wird nun wie folgt berechnet. Zunächst bestimmen wir uns mit Hilfe der Skalen für jede Achse j einen Index i_j, $1 \le j \le d$. Dabei soll in dem Intervall, das durch die neben dem Index i_j liegenden Partitionierungspunkte aufgespannt wird, die Schlüsselkomponente K_j liegen. Somit gilt $K \in S(i_j, j)$. In einem zweiten Schritt wenden wir zu vorgegebenem $m_j = \mid P_j \mid$, $1 \le j \le d$ folgende Adreßfunktion auf das Indexfeld $(i_1, \ldots, i_d)$ an:

$$L_d((i_1, \ldots, i_d), (m_1, \ldots, m_d)) = \sum_{j=1}^{d} i_j \left(\prod_{l=j+1}^{d} (m_l + 1) \right) \tag{2}$$

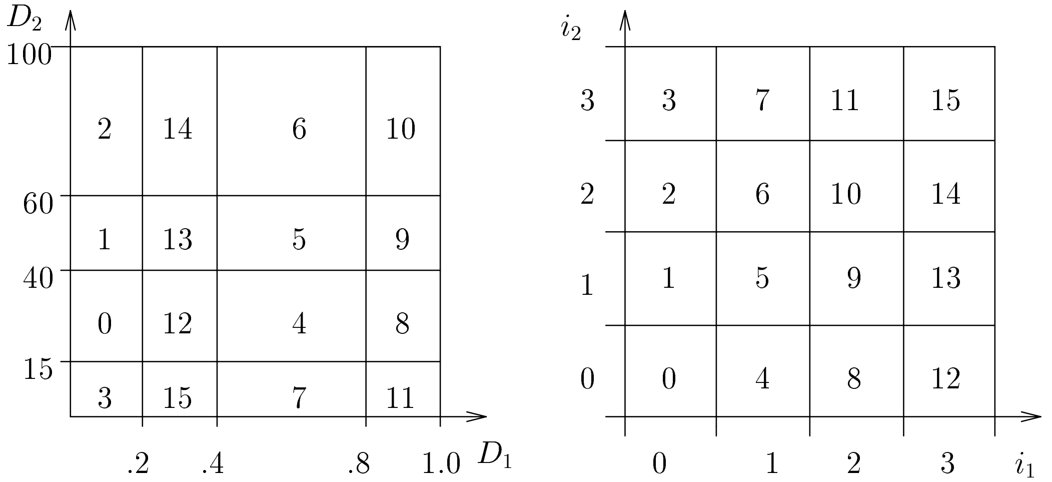


Abbildung 3: Adressen erzeugt durch die LA, wobei die Adressen sowohl in Abhängigkeit der Schlüssel als auch in Abhängigkeit des Indexfeldes (i_1, i_2) dargestellt sind

Als verkürzende Schreibweise führen wir für d-dimensionale Vektoren $(i_1, \ldots, i_d)$ mit ganzzahligen $i_j \geq 0$, $j \in \{1, \ldots, d\}$, die Bezeichnung $\vec{i}$ ein. Seien zwei Vektoren $\vec{i}, \vec{j}$ vorgegeben, so gilt:

$$\begin{aligned} \vec{i} \in \vec{j} \quad &:\Leftrightarrow \quad i_k \in \{0, \ldots, j_k\},\ 1 \leq k \leq d \\ \vec{i} < \vec{j} \quad &:\Leftrightarrow \quad i_k < j_k,\ 1 \leq k \leq d \\ \vec{i} = \vec{j} \quad &:\Leftrightarrow \quad i_k = j_k,\ 1 \leq k \leq d \end{aligned}$$

Weiterhin bezeichnen wir mit $\vec{0}$ den d-dimensionalen Vektor, dessen Komponenten identisch gleich 0 sind.

Wir bezeichnen die Adreßfunktion 2 als lexikographische Adreßfunktion oder kurz mit LA. Die charakteristische Eigenschaft der LA ist, daß die erste Achse dominanter als die zweite Achse ist, die zweite Achse dominanter als die dritte Achse ist, usw. . Dies bedeutet, falls $L_d(\vec{i}, \vec{m}) < L_d(\vec{j}, \vec{m})$ für $\vec{i}, \vec{j} \in \vec{m}$ erfüllt ist, so gilt stets: $i_1 < j_1$, oder $i_1 = j_1$ und $i_2 < j_2$, oder $i_1 = j_1$, $i_2 = j_2$ und $i_3 < j_3$ etc. . Allgemeiner haben wir diesen Sachverhalt in folgendem Lemma formuliert.

Lemma 1
Seien $\vec{m} \geq \vec{0}$ und $\vec{i}, \vec{j} \in \vec{m}$ gegeben. Falls $L_d(\vec{i}, \vec{m}) < L_d(\vec{j}, \vec{m})$ erfüllt ist, gibt es ein $k_0 \in \{1, \ldots, d\}$ mit

$$i_k = j_k \quad \forall\, k \in \{1, \ldots, k_0 - 1\} \quad \wedge \quad i_{k_0} < j_{k_0}$$

Sind aus dem Kontext die Anzahl der Partitionierungspunkte m_i, $1 \leq i \leq d$, bekannt, so werden wir auch die lexikographische Ordnungsrelation $<^L$ benutzen. Hierbei ist für $\vec{i}, \vec{j} \in \vec{m}$ die Ordnungsrelation wie folgt definiert:

$$\vec{i} <^L \vec{j} \;:\Leftrightarrow L_d(\vec{i}, \vec{m}) < L_d(\vec{j}, \vec{m}) \tag{3}$$

Bemerkenswert ist, daß die Adreßfunktion LA unabhängig von m_1, der Anzahl der Partitionierungspunkte in der dominantesten Achse, ist. Dadurch können neue Partitionierungspunkte und neue Scheiben in der dominantesten Achse hinzugefügt werden, ohne

Änderung der Adressen. Würden wir dagegen in anderen Achsen die Anzahl der Partitionierungspunkte erhöhen, so würde die Zuordnung zwischen Indexfeldern und Adressen vollständig geändert werden. Aus diesem Grund ist die Adreßfunktion LA nicht für dynamische Dateien geeignet. Im nächsten Abschnitt werden wir eine dynamische Variante von der LA für PLOP-Hashing vorstellen, die ebenfalls mit Hilfe von Skalen die Adressen berechnet. Zu bemerken ist, daß PLOP-Hashing aus Gründen der Effizienz die Skalen nicht als Listen, sondern als binäre Bäume organisiert.

```
TYPE BinaryTree = POINTER TO Node;
     Node       = RECORD
                    left,right: BinaryTree;
                    CASE leaf: BOOLEAN OF
                        TRUE : index,num: CARDINAL;
                      | FALSE: part     : KeyComponent;
                    END;
                  END;

PROCEDURE Traverse(node: BinaryTree; key: KeyComponent): BinaryTree;
```

Abbildung 4: Spezifikation der binären Bäumen und der Operationen "Traverse"

Die Skalen werden implementierungsunabhängig für eine möglichst gute Anpassung des Gitters an die Verteilung der Daten benötigt. So versucht PLOP-Hashing, auf die jeweiligen Scheiben einer Achse in etwa gleich viele Datensätze zu verteilen. Dies führt zu einem großen Gewinn an Leistung im Vergleich zu den typischen mehrdimensionalen Hashverfahren ohne Directory, wie z. B. dem Interpolationsverfahren [Bur 83], die den Datenraum nur in gleich große Zellen unterteilen können und somit nur gleichverteilte Schlüssel optimal unterstützen.

Im folgenden stellen wir die Struktur der binären Bäume vor, wobei wir sowohl eine informelle Beschreibung als auch die Beschreibung der Struktur in Modula-2 Notation [Wir 85], siehe Abb.4, liefern. Für jede Achse j wird eine Skala und somit ein binärer Baum B_j benötigt, $1 \leq j \leq d$. Wir unterscheiden bei einem binären Baum wie üblich zwischen inneren Knoten und Blättern. Ein innerer Knoten (**leaf = FALSE**) enthält einen Partitionierungspunkt sowie zwei Zeiger zu zwei Teilbäumen, siehe auch Abb.4 und Abb. 5. Die Partitionierungspunkte sind so organisiert, daß alle die kleiner sind als ein Partitionierungspunkt eines beliebigen inneren Knotens, im linken Teilbaum dieses Knotens, alle anderen, die diesem inneren Knoten zugeordnet sind, im rechten Teilbaum abgelegt werden. Ein Blatt eines binären Baums gehört zu genau einer Scheibe S(i_j,j), $0 \leq i_j \leq m_j$, $1 \leq j \leq d$. Deshalb wird in einem Blatt genau der zugehörige Index der Scheibe gehalten. Um ein effizientes sequentielles Abarbeiten von Datensätzen zu erlauben, werden die Blätter miteinander verkettet. Zwei Blätter eines binären Baums sind genau dann miteinander verkettet, falls die zugehörigen Scheiben zueinander benachbart sind. Desweiteren wird eine Zählervariable **num** in einem Blatt abgespeichert, in welcher die Anzahl von Datensätzen in der entsprechenden Scheibe abgelegt wird. Ohne näher auf diese Komponente einzugehen, wird sie für das Erweitern und Verkleinern der Datei

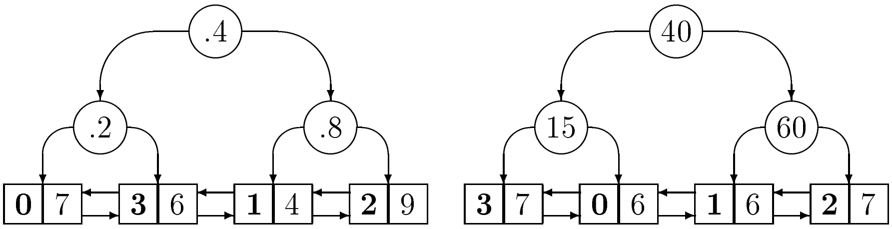


Abbildung 5: Organisation der Skalen durch binäre Bäume

benutzt. Die Datenstruktur eines binären Baums ist hinreichend bekannt. Eine nähere Beschreibung findet sich in einem Buch über Datenstrukturen, siehe z. B. [RH 86]. Trotzdem wollen wir erwähnen, daß wir im folgenden die Funktion **Traverse** benutzen, die zu einer Schlüsselkomponente und einem Knoten (üblicherweise die Wurzel) des zugehörigen Baums, ein Blatt des Baums liefert. Zur weiteren Veranschaulichung der Struktur der binären Bäume haben wir in Abb. 5 die Organisation der Partitionierungspunkte aus unserem Beispiel (siehe Abb. 2) aufgezeigt.

2.2 Die dynamische lexikographische Adreßfunktion

In diesem Abschnitt werden wir die dynamische Adreßfunktion oder genauer gesagt die Folge von Hashfunktionen beschreiben, die zur Adressierung der Ketten bei PLOP-Hashing benutzt wird. Eine ähnliche Adreßfunktion wurde in [Oto 84] für die Adressierung der Adreßtabelle von mehrdimensionalen erweiterbaren Hashing vorgestellt. Ähnlich zu der LA besteht die Adreßberechnung aus zwei Schritten. In dem ersten Schritt wird für jede Achse j mittels der binären Bäume ein Index i_j, $1 \leq j \leq d$ bestimmt. Mit Hilfe zusätzlicher Informationen erhalten wir durch eine Hashfunktion $H_{d,m} : (i_1, \ldots, i_d) \mapsto \{0, \ldots, m-1\}$ unsere gewünschte Adresse. Bevor wir eine detaillierte Beschreibung liefern, führen wir einige neue Begriffe ein.

Das Level L der Datei gibt, wie bei linearem Hashing, die Anzahl der Verdopplungen der Dateigröße m an. Dabei ist die Datei stets durch eine Kette initialisiert, die genau aus einer leeren Seite besteht. Somit ergibt sich L durch $L = \lfloor \log_2 m \rfloor$. Ohne bereits hier die dynamische Organisation von PLOP-Hashing erklären zu wollen, fordert PLOP-Hashing für jedes Level l, $0 \leq l \leq L$ eine sogenannte Splitachse s, $1 \leq s \leq d$. Dadurch ist angezeigt, daß das Anwachsen der Datei von 2^l auf $2^{l+1} - 1$ Ketten durch Erweiterung bzgl. der s-ten Achse realisiert wurde, d. h. die Gitterpartitionierung $G_D(\overline{P}_1, \ldots, \overline{P}_d)$ wird durch Vergrößerung der Menge $\overline{P}_s$ verfeinert. Im Gegensatz zu anderen mehrdimensionalen Hashverfahren ist die Wahl, welche Achse zur nächsten Splitachse bestimmt wird, bei PLOP-Hashing nicht festgelegt. Im Extremfall könnte der Benutzer selbst interaktiv die nächste Splitachse angeben. Daher benötigt PLOP-Hashing ein Feld $H = (H_0, \ldots, H_L)$, in welchem die Splitgeschichte abgelegt ist. Die i-te Komponente H_i, $i \leq L$, entspricht dabei der ehemaligen Splitachse als die Datei zwischen 2^i und $2^{i+1} - 1$ Ketten besaß.

Ähnlich zu der LA benötigt PLOP-Hashing für seine dynamische Adreßfunktion die Anzahl der Partitionierungspunkte m_j in der j-ten Achse, $1 \leq j \leq d$.

Für eine intuitive Vorstellung der Adreßfunktion betrachten wir für d=2 ein Beispiel.

In Abb. 6 haben wir hierzu die Abhängigkeit der Adressen von dem Indexfeld $\vec{i} \in (7, 9)$ aufgezeigt. Hierbei fällt auf, daß für $\vec{i} \in (3, 3)$ die selben Kettenadressen wie für die LA erzeugt wurden. Dies ergibt sich durch die spezielle Wahl der ersten vier Splitachsen. Die nächsten 16 Adressen $16, \ldots, 31$ sind nun nicht mehr durch die statische LA berechenbar. Trotzdem ist zu erkennen, daß diese Adressen in lexikographischer Reihenfolge angeordnet sind, wobei die zweite Achse die dominante Achse ist. Nach einer weiteren Verdopplung wurden die Ketten mit Adressen $32, \ldots, 63$ angefügt. Wiederum sind die Adressen lexikographisch sortiert, wobei nun wieder die erste Achse dominant zur zweiten Achse ist. Für die weiteren Achsen $64, \ldots, 79$ ist nun wiederum die zweite Achse die dominante Achse. Die dominante Achse der Adressen $2^l, \ldots, 2^{l+1} - 1$ ergibt sich nun gerade aus der zugehörigen Splitachse, welche der l-ten Komponente H_l der Splitgeschichte entspricht, $0 \leq l < L$. Für die aktuelle Splitachse $s = H_L$ ist es nicht notwendig, daß bereits alle Adressen $2^L, \ldots, 2^{L+1} - 1$ in der Datei existieren.

Seien nun die Werte von d, L, $\vec{m}$, s und H=$(H_0, \ldots, H_L)$ gegeben, so wird die Adresse zu dem Indexfeld $\vec{i} \in \vec{m}$ so berechnet, daß die frühere Situation der Datei, als dieses Indexfeld zum ersten Mal benutzt wurde, rekonstruiert wird. Insbesondere wird das ehemalige Level pL der Datei, die ehemalige Anzahl von Partitionierungspunkte $\vec{pm} \in \vec{m}$ in den einzelnen Achsen und die ehemalige Splitachse ps (d. h. die dominante Achse bzgl. der lexikographischen Ordnung) berechnet. Aus diesen Werten läßt sich dann die jeweilige Adresse bestimmen. Die genaue Beschreibung, wie diese Werte und schließlich die Adresse bestimmt werden, ist in dem Algorithmus DLA gegeben.

i_2									
9	72	73	74	75	76	77	78	79	
8	64	65	66	67	68	69	70	71	
7	28	29	30	31	39	47	55	63	
6	24	25	26	27	3 8	46	54	62	
5	20	21	22	23	37	45	53	61	
4	16	17	18	19	36	44	52	60	
3	3	7	11	15	35	43	51	59	
2	2	6	10	14	34	42	50	58	
1	1	5	9	13	33	41	49	57	
0	0	4	8	12	32	40	48	56	
	0	1	2	3	4	5	6	7	i_1

Abbildung 6: Beispiel für die Adressierung der dynamischen lexikographischen Adreßfunktion, wobei $H = (2, 2, 1, 1, 2, 1, 2), s = 2, \vec{m} = (7, 9), L = 6, d = 2$

Algorithmus DLA($\vec{i}$,$\vec{m}$)
gegeben: eine mittels PLOP-Hashing organisierte Datei, Indexfeld $\vec{i} \in \vec{m}$

```
1. IF $\vec{i} = \vec{0}$ THEN
         RETURN 0;
   END;
2. (* Berechnet die "ehemalige" Splitachse ps und das "ehemalige" Level pL *)
   FOR j := 1 TO d DO
         IF $i_j = 0$ THEN
            $h_j := 0$
         ELSE
            $l_j := \lfloor \log_2 i_j \rfloor + 1$;
            Bestimme $h_j$ so daß folgende Bedingungen erfüllt sind:
            a) $H[h_j] = j$
            b) $l_j = | \{ i \mid H[i] = j, 0 \leq i \leq h_j \} |$
         END;
   END;
   pL := $\max_{1 \leq j \leq d} h_j$;
   ps := H[pL];
3. (* Berechnet die Anzahl von Zellen in einer Scheibe *)
   FOR j := 1 TO d DO
         c := $| \{ i \mid H[i] = j, 0 \leq i < pL \} |$;
         $pm_j := 2^c$
   END;
4. (* Brechnung der endgültigen Adresse *)
   base := $2^{pL}$;
   $\vec{op}_1 := (i_{ps} - pm_{ps}, i_1, \ldots, i_{ps-1}, i_{ps+1}, \ldots, i_d)$;
   $\vec{op}_2 := (pm_{ps}, pm_1, \ldots, pm_{ps-1}, pm_{ps+1}, \ldots, pm_d)$;
   offset := $L_d(\vec{op}_1, \vec{op}_2)$
5. RETURN (base + offset);
```

Wie in dem Algorithmus DLA zu sehen ist, wird in Schritt 4 die statische LA benutzt, wobei die erste Komponente der Eingabevektoren zu der ehemaligen Splitachse ps zugeordnet ist und somit die ps-te Achse dominant zu den anderen Achsen ist. Man beachte, daß $pm_{ps} = 2^{\lfloor \log_2 i_{ps} \rfloor}$ und daher $pm_{ps} \leq i_{ps}$ gilt.

Zu einem vorgegebenen $\vec{i} \in \vec{m}$ beträgt der Aufwand zur Berechnung der zugehörigen Adresse durch den Algorithmus DLA O(d). Im wesentlichen besteht der Algorithmus aus den zwei FOR-Schleifen in den Schritten 2 und 3, die jeweils d-mal durchlaufen werden und dem Aufruf der statischen lexikographischen Adreßfunktion in Schritt 4. Für die statische LA werden ebenfalls O(d) Rechenoperationen benötigt. Der Aufwand einen Schlüssel K in ein Indexfeld umzuwandeln, beträgt unter der Voraussetzung von balancierten binären Bäumen $O(\log\ n)$, so daß wir eine Adresse in logarithmischer Zeit berechnen können. Dies ist vergleichbar zu anderen Adreßfunktionen wie z.B der Interpolationsfunktion [Bur 83].

Betrachten wir unser Beispiel aus Abb. 6, so ergeben sich für die Indexfelder (7,4) und (3,5) folgende Werte der Variablen in den einzelnen Schritten des Algorithmus DLA:

	Indexfeld (7,4)	Indexfeld (3,5)
2.	$l_1 = 3$, $l_2 = 3$	$l_1 = 2$, $l_2 = 3$
	$h_1 = 5$, $h_2 = 4 \Rightarrow pL = 5 \wedge ps = 1$	$h_1 = 3$, $h_2 = 4 \Rightarrow pL = 4 \wedge ps = 2$
3.	$pm_1 = 4$, $pm_2 = 8$	$pm_1 = 4$, $pm_2 = 4$
4.	base = 32	base = 16
	offset = L_2((3,4), (4,8)) = 28	offset = L_2((1,3), (4,4)) = 7
5.	Resultat: 60	Resultat: 23

2.3 Dynamische Organisation der Datei

In vielen Anwendungen werden sogenannte dynamische PZS benötigt, deren Effizienz unabhängig von der Anzahl der Datensätze ist. Hierzu führen dynamische PZS lokale Reorganisationen durch, die maximal folgendes leisten sollten:

- Anhängen (weniger) leerer Seiten an die Datei und Umspeicherung in der Datei liegender Datensätze in die neuen Seiten

- Umspeichern aller Datensätze aus einer begrenzten Anzahl von Seiten in andere Seiten der Datei und Freigabe der leeren Seiten

- Neuverteilung aller Datensätze aus einer begrenzten Anzahl von Seiten auf diese Seiten

Im Gegensatz zu einer lokalen, wird bei einer globalen Reorganisation einer Datei auf (fast) alle Seiten der Datei zugegriffen. Lokale Reorganisationen werden durch eine Einfüge- bzw. Löschoperation eines Benutzers ausgelöst, während globale Reorganisationen i. a. vom Systembetreuer initiiert werden. Der kritische Aspekt bei Reorganisationen ist, daß während ihres Ablaufs alle extern dem Benutzer zur Verfügung stehenden Operationen auf dem zu reorganisierenden Datenbestand gesperrt sind. Speziell für sehr große Dateien ist deshalb die Forderung nach dynamischen PZS und sehr kurzen lokalen Reorganisationen unverzichtbar.

PLOP-Hashing benutzt für die lokale Reorganisation der Datei, unter Beachtung obiger Anforderungen, drei interne dem Benutzer nicht direkt zur Verfügung stehenden, Operationen. Neben dem Splitten und Verschmelzen von Datenketten, die in jeder dynamischen PZS vorzufinden sind, besitzt PLOP-Hashing eine dritte Operation zum Anpassen an die Verteilung der Datensätze. Für jede dieser internen Operationen gibt es eine Kontrollfunktion, die den Status der Datei überprüft und gegebenenfalls die zugehörige interne Operation aufruft. Typischerweise werden die Kontrollfunktionen nach dem Löschen und Einfügen von Datensätzen betrachtet.

Für die Kontrollfunktionen von PLOP-Hashing benötigen wir den Belegungsfaktor lf(i,j) der Scheiben S(i,j), der durch das Verhältnis der Anzahl der Datensätze in der Scheibe S(i,j) zu der maximalen Anzahl von Datensätzen, die in den Primärseiten der Scheibe S(i,j), $0 \leq i \leq m_j$, $1 \leq j \leq d$, Platz finden, gegeben ist.

2.3.1 Erweitern

Sei eine mittels PLOP-Hashing organisierte Datei gegeben. Somit sind entsprechende Parameter, wie d, s, $P_i, \ldots$, wohl definiert. Nehmen wir an, daß die Kontrollfunktion ein Erweitern der Datei fordert, so gehen wir in folgenden Schritten vor:

1. Bestimme j_0 so, daß lf(j_0,s) = $\max_{0 \leq j \leq m_s}$ lf(j,s)

2. Berechne die begrenzenden Partitionierungspunkte $A, B \in \overline{P}_s$ der Scheibe S(j_0,s) und setze AB = $(A + B)/2$

3. Füge den neuen Partitionierungspunkt AB in den binären Baum der s-ten Achse ein ($P_s := P_s \cup \{AB\}$) und setze $m_s :=\mid P_s \mid$. Weise dem neu erzeugten Blatt den Index m_s zu.

4. Splitte alle Ketten, die zur Scheibe S(j_0,s) gehören bzgl. der s-ten Schlüsselkomponente und dem neuen Partitionierungspunkt AB.

Betrachten wir hierzu für d=2 und s=1 unser Beispiel aus Abb. 2 (siehe S. 17) in Kombination mit den binären Bäumen aus Abb. 5. Im ersten Schritt erhalten wir $j_0 = 2$, da die Scheibe S(2,1) den höchsten Belegungsfaktor besitzt. Die Scheibe S(2,1) wird durch die Partitionierungspunkte 0.8 und 1.0 begrenzt. Somit erhalten wir AB = 0.9 als neuen Partitionierungspunkt. Sodann passen wir den binären Baum der ersten Achse an. Schließlich splitten wir die vier Ketten der Scheibe S(2,1) bzgl. des neuen Partitionierungspunkts in 8 Ketten auf. Wir erhalten somit die Situation, wie sie in Abb. 7 veranschaulicht ist.

Beim Einfügen einer neuen Hyperebene in das Gitter entstehen aus einer Scheibe zwei neue Scheiben. Die Adressen der Ketten der neuen linken Scheiben übernehmen die Adressen der Ketten von der ursprünglichen Scheibe. Die Primärseiten der Ketten von der rechten Scheibe sind am Ende der Primärdatei zu finden und belegen somit die höchsten Adressen in der Datei. In unserem Beispiel aus Abb. 7 erhalten somit die Ketten der Scheibe S(4,1) die Adressen $16, \ldots, 19$.

Die im letzten Abschnitt erwähnte Splitachse wird stets dann neu gewählt, wenn die Datei ein neues Level erreicht hat. Diese Splitachse bleibt fest bis sich die Dateigröße erneut verdoppelt hat. Um eine Adressierung zu ermöglichen, müssen die Splitachsen in einem Feld abgelegt werden, das wir als Splitgeschichte bezeichnen. Andere PZS, wie das Interpolationsverfahren [Bur 83] setzen meist eine feste Splitreihenfolge voraus, so daß jede Achse etwa dieselbe Anzahl von Partitionierungspunkten besitzt. Wir möchten hier bemerken, daß eine mehrdimensionale PZS eine beliebige Gewichtung der Achsen zulassen muß, so daß insbesondere partielle Anfragen (beliebiger Verteilung) effizient unterstützt werden können. Das Problem, für entsprechende Anfrageverteilungen von partiellen exakten Anfragen die optimale PZS zu finden, ist in der Literatur ausführlich diskutiert worden [LR 82].

Die von PLOP-Hashing benötigte Kontrollfunktion legt den Zeitpunkt für die Erweiterung der Datei fest. Sie wird dabei nach jeder Einfügeoperation überprüft. Zunächst betrachten wir folgende Kontrollfunktion:

(Ex α) Bestimme in der aktuellen Splitachse s die Scheibe S(i,s), $0 \leq i \leq m_s$, die die maximale Anzahl von Datensätzen enthält. Falls der Belegungsfaktor dieser

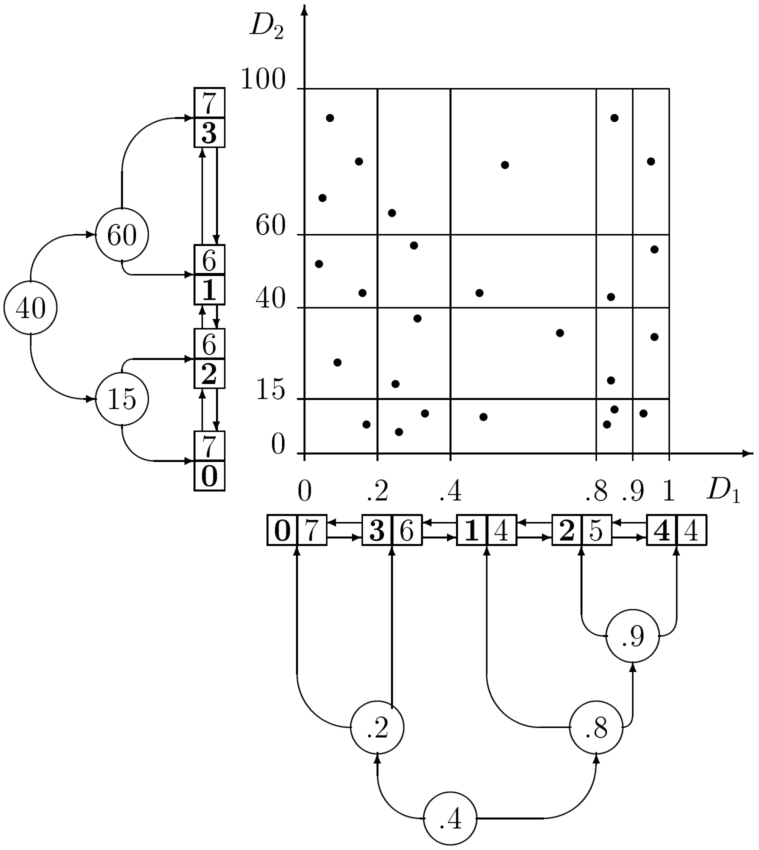


Abbildung 7: Erweiterung der Datei

Scheibe größer als α ist, wird eine Erweiterung der Datei, genauer gesagt der Scheibe, ausgelöst.

Zunächst wählen wir den Wert $\alpha = 1$. Unter der Annahme, daß ständig Datensätze eingefügt werden, bewirkt eine Vergrößerung von α eine verzögerte Ausführung der Splitoperationen. Dadurch ergeben sich längere Ketten, höhere Kosten bei einer exakten Suche, aber auch eine höhere Span. Deshalb ist es durchaus möglich, daß Bereichsanfragen durch höhere Werte von α besser unterstützt werden. Die Wahl des optimalen Werts für α hängt direkt von der durchschnittlichen Größe der Anfrageregionen ab.

Bei früheren MDH ohne Directory ist während des Anwachsens einer Datei von 2^l auf $2^{l+1}-1$ Ketten, jede Kette mit Adresse j, $0 \leq j < 2^l$, $l > 0$, in zwei Ketten aufgeteilt worden. Dagegen können bei PLOP-Hashing, während einer Verdopplung der Datei, Ketten sowohl überhaupt nicht als auch mehrfach gesplittet werden. So würde z. B. in Abb. 7 die Scheibe S(1,1) nicht erweitert werden, falls alle neu eingefügten Datensätze außerhalb der Scheibe liegen. Dagegen würden genügend viele Einfügungen in der Scheibe S(2,1) ein erneutes Erweitern dieser Scheibe auslösen. Die Auswahl, welche Scheibe gesplittet wird, hängt bei PLOP-Hashing nur von der Belegung der jeweiligen Scheibe in der Splitachse ab (siehe Schritt 1). Deshalb trägt diese Kontrollfunktion auch dazu bei, daß die Gitterpartitionierung sich an die jeweilige Verteilung der Daten anpaßt.

Betrachten wir nun näher wie PLOP-Hashing das Splitten der Ketten realisiert, nachdem eine Scheibe durch die Kontrollfunktion (Ex 1) bestimmt wurde. Da die Belegung in dieser Scheibe sehr hoch ist, sollte das Erweitern der zugehörigen Ketten so schnell wie möglich ablaufen. Ein erster Vorschlag könnte sein, alle Ketten in einem Reorganisationsschritt zu splitten. Dies erfordert viele Diskzugriffe und würde deshalb für einen längeren Zeitraum Operationen des Benutzers auf der Datei nicht erlauben. Dieser Vorschlag berücksichtigt nicht die Forderung, daß die Antwortzeit einer Operation nicht entscheidend von Reorganisationen der Datei beeinflußt werden sollte.

Unser Vorschlag ist, das Splitten von Scheiben in mehrere Schritte aufzuteilen, wobei genau ein Schritt pro Einfüge- oder Entferneoperation ausgeführt wird. Die zu splittende Scheibe kann man sich dabei als eine eigene Datei vorstellen, die, wie bei linearem Hashing, linear erweitert wird. Dementsprechend benötigt PLOP-Hashing einen Expansionszeiger, der auf die nächste zu splittende Kette in der Scheibe verweist. Dieser Zeiger ermöglicht die Entscheidung, ob eine Kette in der Scheibe bereits gesplittet wurde oder noch zu splitten ist. Da die Adreßfunktion DLA von PLOP-Hashing auf der lexikographischen Ordnung basiert, wird auch eine Scheibe linear bzgl. der lexikographischen Ordnung erweitert.

Der Expansionszeiger entspricht einem d-dimensionalen Feld $\vec{ep} = (ep_1, \ldots, ep_d) \in \vec{m}$, wobei DLA$(\vec{ep}, \vec{m})$ die Adresse der Kette ist, die bei der nächsten Einfüge- oder Löschoperation gesplittet wird. Während einer Erweiterung teilen wir genau die Kette mit Adresse DLA$(\vec{ep}, \vec{m})$ in zwei Ketten auf, wobei die eine Kette (in der rechten Scheibe) die neue Adresse DLA$(ep_1, \ldots, ep_{s-1}, m_s, ep_{s+1}, \ldots, ep_d, \vec{m})$ und die andere Kette die Adresse DLA$(\vec{ep}, \vec{m})$ zugewiesen bekommt. Diese neue Adresse entspricht gerade der Position am Ende der Datei, so daß der Adreßraum kompakt bleibt. Diese Kompaktheit kann nur durch eine geeignete Berechnung des neuen Werts von ep garantiert werden.

Nachdem die Kontrollfunktion ein Splitten der Scheibe S(i_s,s) fordert, wird der Expansionszeiger $\vec{ep}$ wie folgt initialisiert:

$$ep_j := \begin{cases} 0 & \text{falls } j \neq s \\ i_s & \text{sonst} \end{cases} \qquad j \in \{1, \ldots, d\}$$

Mit Ausnahme des letzten Expansionsschrittes, wird nach dessen Ausführung der Expansionszeiger $\vec{ep}$ neu angepaßt. Dabei wird nur der (d-1)-dimensionale Teilvektor $(ep_1, \ldots, ep_{s-1}, ep_{s+1}, .$ abgeändert, während die s-te Komponente ep_s unverändert bleibt. Ist die ganzzahlige Zahl c, $c \geq 0$, durch

$$c := L_{d-1}((ep_1, \ldots, ep_{s-1}, ep_{s+1}, \ldots, ep_d), (m_1, \ldots, m_{s-1}, m_{s+1}, \ldots, m_d))$$

gegeben, so ist der neue Teilvektor durch

$$L_{d-1}((ep_1, \ldots, ep_{s-1}, ep_{s+1}, \ldots, ep_d), (m_1, \ldots, m_{s-1}, m_{s+1}, \ldots, m_d)) := c + 1$$

bestimmt. D.h. der (d-1)-dimensionale Vektor $(ep_1, \ldots, ep_{s-1}, ep_{s+1}, \ldots, ep_d)$ wird auf den lexikographisch nächsthöheren Wert gesetzt.

Lemma 2
Sei s die aktuelle Splitachse und S(i_s,s) die Scheibe in welcher sich die zu splittenden Ketten befinden. Falls $\vec{ep}$ wie oben beschrieben berechnet wird, verweist die Adresse DLA($ep_1, \ldots, ep_{s-1}, m_s, ep_{s+1}, \ldots, ep_d, \vec{m}$) stets auf das Ende der Datei, d. h. es gibt keine größere Adresse in der Datei.

Der Beweis dieses Lemmas ergibt sich aus der Beobachtung, daß die Ketten in der neuen Scheibe S(m_s,s) die höchsten Adressen der Datei besitzen, siehe Algorithmus DLA. Innerhalb dieser Scheibe werden die Adressen im wesentlichen durch die statische lexikographische Adreßfunktion bestimmt. Insbesondere gilt:

$$\begin{aligned} & DLA((i_1, \ldots, i_{s-1}, m_s, i_{s+1}, \ldots, i_d), \vec{m}) < DLA((j_1, \ldots, j_{s-1}, m_s, j_{s+1}, \ldots, j_d), \vec{m}) \\ \iff & (i_1, \ldots, i_{s-1}, i_{s+1}, \ldots, i_d) <^L (j_1, \ldots, j_{s-1}, j_{s+1}, \ldots, j_d) \\ & \forall i_k, j_k : 0 \leq i_k, j_k \leq m_k,\ k \in \{1, \ldots, d\} \setminus \{s\} \end{aligned}$$

Ergänzend zum Erweitern der Datei muß noch erwähnt werden, daß der binäre Baum der Splitachse erst nach dem Splitvorgang für eine Scheibe angepaßt wird.

Unsere Anforderung an das Erweitern der Datei, nämlich das Splitten einer Scheibe möglichst schnell zu vollziehen ohne dabei eine Einfüge- oder Löschoperation überdurchschnittlich zu belasten, ist somit im wesentlichen erfüllt. Die Anzahl der Diskzugriffe für einen Erweiterungsschritt ist dabei beschränkt durch die Länge der zu splittenden Kette. Zu beachten ist, daß selbst nach dem Löschen eines Datensatzes aus der Datei ein Erweitern der Datei stattfinden kann.

Wir möchten hier auch kurz auf den Namen des Verfahrens eingehen. PLOP-Hashing steht für "piecewise linear order preserving hashing". Dieser Name ergibt sich dadurch, daß die Datei in Stücke aufgeteilt wird und innerhalb dieser Stücke linear erweitert wird. Die Stücke der Datei entsprechen dabei genau den Scheiben in der aktuellen Splitachse.

2.3.2 Verkleinern

Neben dem Erweitern ist das Verkleinern einer Datei eine wichtige interne Operation jeder PZS. Zum einen soll für PLOP-Hashing durch das Verkleinern die Leistung erhalten bleiben, falls viele Datensätze gelöscht worden sind, und zum anderen kann dadurch wieder die Gitterpartitionierung an die Verteilung der Datensätze angepaßt werden. Im Prinzip ist das Verkleinern die inverse Operation zum Erweitern der Datei.

Entsprechend dem Erweitern betrachten wir eine spezielle Achse, in welcher das Verschmelzen von Scheiben erlaubt ist. Wir bezeichnen die Achse im folgenden als Mischachse. Die Mischachse entspricht fast immer der Expansionsachse mit der Ausnahme, in welcher die Datei aus 2^l Ketten besteht, $l > 0$. In solch einer Situation ergibt sich die Mischachse aus der früheren Expansionsachse, die der $(l-1)$-ten Komponente der Splitgeschichte H entspricht.

Um den Zeitpunkt für das Verkleinern festzulegen, benötigen wir eine Kontrollfunktion, die wie folgt gegeben ist:

(Co β) Bestimme in der Mischachse das Paar von Scheiben mit der minimalen Anzahl von Datensätzen. Falls die Belegung dieser Scheiben unter β liegt, sollen diese Scheiben zu einer Scheibe verschmolzen werden.

Damit, nachdem gerade eine Scheibe gesplittet wurde, die so erzeugten Scheiben im darauf folgenden Schritt nicht wieder miteinander verschmolzen werden müssen, muß $\beta < \alpha/2$ erfüllt sein, wobei $\alpha > 0$ durch die Kontrollfunktion (Ex α) fürs Splitten bestimmt ist. Für $\alpha = 1$ haben wir im folgenden beispielhaft den Wert $\beta = 0.45$ gewählt.

Entsprechend zum Erweitern wird ein Verschmelzen von zwei benachbarten Scheiben in mehreren Schritten ausgeführt, wobei in einem Schritt genau zwei benachbarte Ketten miteinander verschmolzen werden. Ein Schritt wird dabei nach einer Einfüge- oder Löschoperation ausgeführt. Ein Zeiger, den wir auch mit $\vec{ep}$ bezeichnen, hält fest, welche Ketten in den Scheiben bereits verschmolzen sind und welche noch zu verschmelzen sind. Wird ein Verschmelzen von zwei Scheiben S(l_s,s) und S(r_s,s) gefordert, so wird zunächst der binäre Baum an die Situation angepaßt, als wäre das Verschmelzen bereits vollzogen. Danach wird der Zeiger $\vec{ep}$ wie folgt initialisiert:

$$ep_j := \begin{cases} m_j & \text{falls } j \neq s \\ l_s & \text{sonst} \end{cases} \qquad j \in \{1, \ldots, d\}$$

Mit der Ausnahme der ersten beiden Seiten wird vor dem Verschmelzen von Seiten, der Zeiger $\vec{ep}$ neu angepaßt. Analog zum Splitten wird nur der (d-1)-dimensionale Teilvektor $(ep_1, \ldots, ep_{s-1}, ep_{s+1}, \ldots, ep_d)$ neu berechnet. Gilt dabei

$$c := L_{d-1}((ep_1, \ldots, ep_{s-1}, ep_{s+1}, \ldots, ep_d), (m_1, \ldots, m_{s-1}, m_{s+1}, \ldots, m_d))$$

so ist der neue Teilvektor durch

$$L_{d-1}((ep_1, \ldots, ep_{s-1}, ep_{s+1}, \ldots, ep_d), (m_1, \ldots, m_{s-1}, m_{s+1}, \ldots, m_d)) := c - 1$$

gegeben. Danach werden die Seiten DLA($(ep_1, \ldots, ep_{s-1}, r_s, ep_{s+1}, \ldots, ep_d), \vec{m}$) und DLA($\vec{ep}, \vec{m}$) miteinander verschmolzen.

Das Verschmelzen von Ketten zweier beliebig benachbarter Scheiben kann nicht durch die inverse Anwendung des Splitalgorithmus erfolgen, da nicht notwendigerweise die Scheibe mit der höchsten Adresse beim Verschmelzen beteiligt ist und somit ein inkompakter Adreßraum entstehen kann. Diese Situation haben wir an Hand eines Beispiels in Abb. 8 veranschaulicht, wo ein Verschmelzen der Scheiben S(1,1) und S(3,1) von der Kontrollfunktion (Co 0.45) gefordert wird. Wie zu sehen ist, besitzt die Scheibe S(4,1) die Ketten mit den höchsten Adressen in der gesamten Datei. Im ersten Schritt der Verschmelze-Operation werden die Ketten mit Adressen 13 und 15 miteinander verschmolzen, wobei die resultierende Kette die Adresse 15 zugewiessen bekommt. Der Platz der Primärseite mit Adresse 13 in der Primärdatei wird zwar nicht mehr benötigt, die Seite kann aber nicht an das Betriebssystem zurückgegeben werden, da in der Datei Seiten mit höheren Adressen existieren. Die Folge wäre eine entsprechend geringe Speicherplatzausnutzung. Um dies zu verhindern, wird die Primärseite von der Kette mit der größten Adresse auf diese nicht benötigte Seite in der Primärdatei kopiert und danach die Seite am Ende der Primärdatei freigegeben. In unserem Beispiel aus Abb. 8 wird also der Inhalt der Primärseite 19 auf die Adresse 13 kopiert und anschließend die Primärseite 19 an das Betriebssystem zurückgegeben. Im nächsten Schritt des Verschmelzens der Scheiben werden die Ketten mit Adressen 9 und 11 verschmolzen, der Inhalt der Primärseite mit Adresse 18 auf die Adresse 9 kopiert und schließlich wird die Primärseite mit Adresse 18 freigegeben.

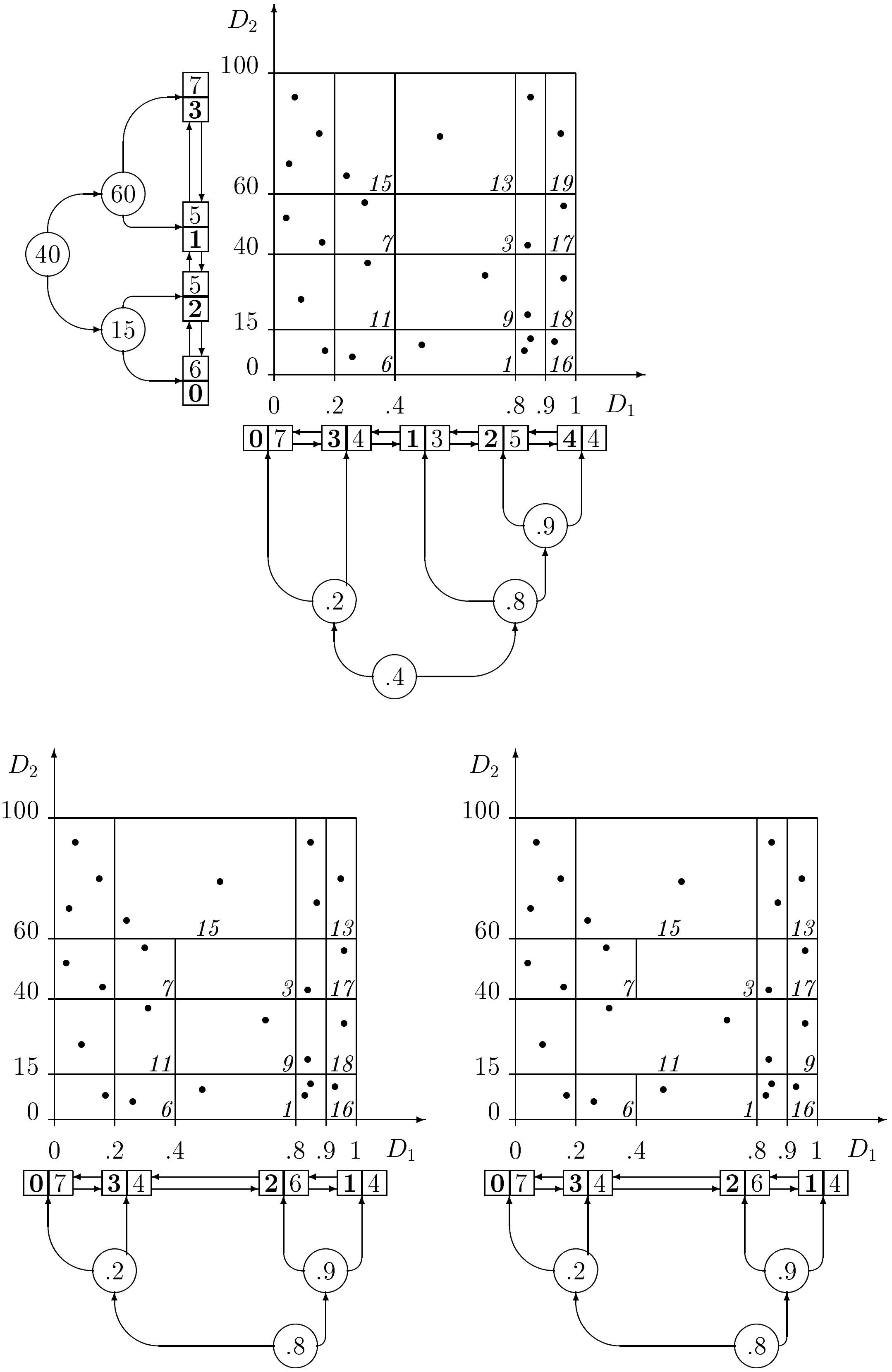


Abbildung 8: Verschmelzen von Ketten, deren Adressen in den zugehörigen Zellen des Gitters zu finden sind

Bevor die eigentliche Reorganisation beginnt und Datensätze umgespeichert werden, wird der binäre Baum in der Mischachse angepaßt. In unserem Beispiel werden die beiden Blätter der entsprechenden Scheiben miteinander verschmolzen, der zugehörige Partitionierungspunkt wird aus dem Baum entfernt und der (höchste) Index 4 durch den Index 1 überschrieben.

2.3.3 Anpassen der Partitionierung an die Verteilung

Das Erweitern und Verkleinern sind interne Operationen von PLOP-Hashing, die für die dynamische Organisation der Datei verantwortlich sind. Wie bereits erwähnt tragen sie auch zur Anpassung der Gitterpartitionierung an die Verteilung bei. PLOP-Hashing benötigt hierfür eine weitere interne Operation, die wir als Anpassen bezeichnen. Betrachten wir hierzu eine Datei, die mittels PLOP-Hashing organisiert wird. Durch mehrfaches Ausführen der Operationen Einfügen und Löschen eines Datensatzes kann die Anzahl der Datensätze konstant gehalten werden. Die Verteilung der Daten kann sich jedoch stark verändern. Dadurch können in einigen Scheiben nur noch wenige Datensätze liegen. Andererseits können diese nahezu leeren Scheiben durch Verkleinern der Datei i. a. nicht beseitigt werden, da die Mischachse nicht der Achse entspricht, die den leeren Scheiben zugeordnet ist. Da PLOP-Hashing während einer Verdopplung einer Datei keinen Einfluß auf die Wahl der Expansionsachse und damit auch auf die Wahl der Mischachse besitzt, ist nicht dafür gesorgt, daß durch das Erweitern und Verkleinern der Datei eine genügend gute Anpassung an die Verteilung erfolgt. PLOP-Hashing garantiert nur in der Expansionsachse, bzw. Mischachse, eine gleichmäßige Verteilung der Datensätze auf die Scheiben.

Unser Vorschlag ist, sich für jede Achse, die nicht Expansion- oder Mischachse ist, zum einen die Scheibe zu merken, die die höchste Belegung besitzt und zum anderen das Paar von benachbarten Scheiben zu merken, das die niedrigste Belegung besitzt. Eine Anpassung der Datei wird für $\gamma > 0$ durch folgende Kontrollfunktion ausgelöst:

(Acc γ) Sei $j \in \{1, \ldots, d\}$ nicht die Expansions- oder Mischachse, max der maximale Belegungsfaktor einer Scheibe und min der minimale Belegungsfaktor eines Paares benachbarter Scheiben in der j-ten Achse. Falls $max > 2 * (min + \gamma)$ ist, soll, bei gleichzeitigem Verschmelzen der beiden minimal belegten Scheiben, die maximal belegte Scheibe gesplittet werden.

Wie in der Kontrollfunktion bereits angezeigt wird, besteht das Anpassen der Datei einmal aus dem Splitten einer Scheibe S(i,j) und dem Verschmelzen eines Paares von Scheiben S(f,j) und S(g,j), $0 \leq f, g, i \leq m_j$, wobei j die entsprechende Achse ist, in welcher die Anpassung durchgeführt wird. Die Anpassung wird analog zum Erweitern und Verkleinern der Datei schrittweise durchgeführt. Bei jedem Schritt betrachten wir dabei drei Ketten, einmal die Kette adr_i, die zur Scheibe S(i,j) gehört, und zum anderen die Ketten adr_f und adr_g, die entsprechend zu den Scheiben S(f,j) und S(g,j) gehören. Wichtig ist, daß diese Ketten in genau (d-1) Scheiben gemeinsam liegen. Sodann werden die Ketten adr_f und adr_g miteinander verschmolzen und die neu entstandene Kette erhält die Adresse adr_f. Schließlich wird die Kette adr_i gesplittet, wobei die neue linke Kette die Adresse der ursprünglichen Kette übernimmt und die rechte Kette die Adresse adr_g zugewiesen bekommt. Ein Zeiger $\vec{ep}$ verweist stets auf das nächste Tripel von Ketten, die,

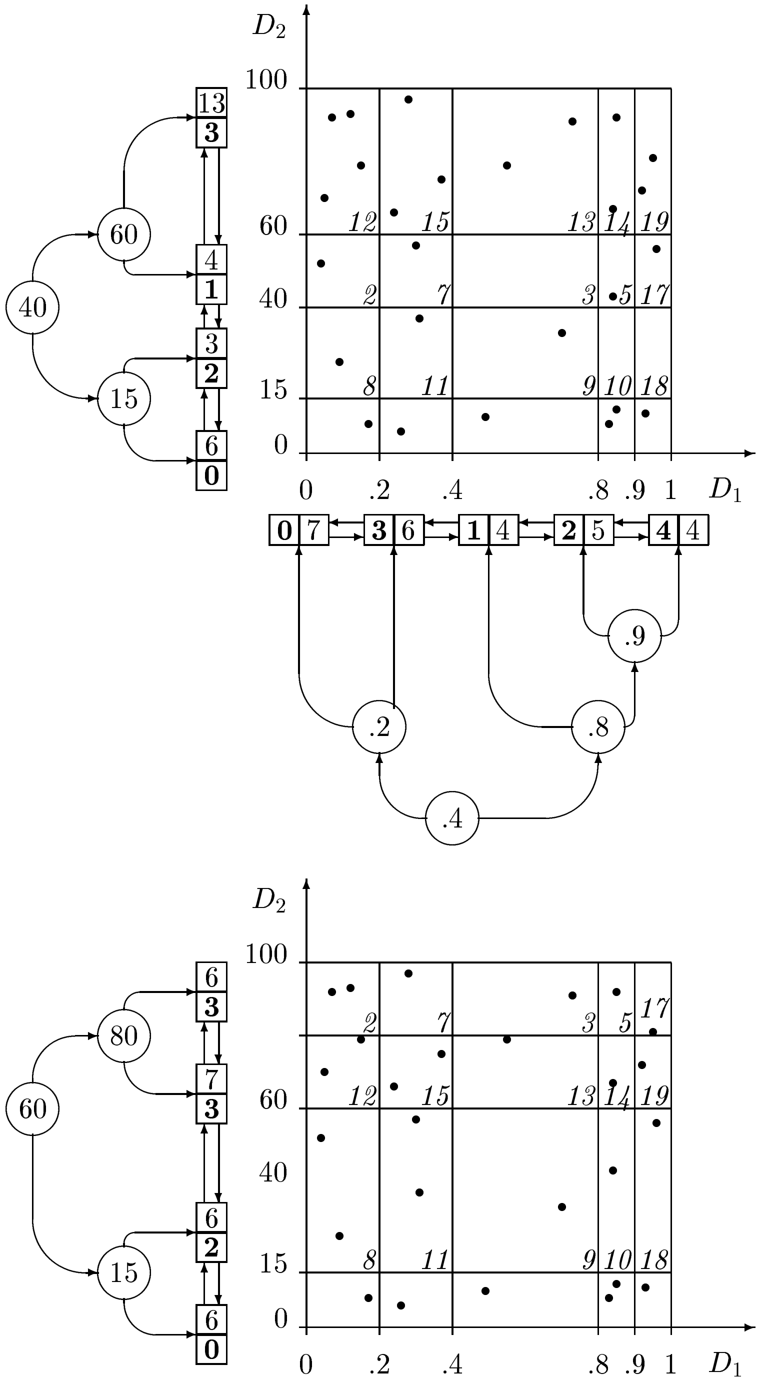


Abbildung 9: Anpassen der Gitterpartitionierung an die Verteilung

wie oben beschrieben, im nächsten Schritt reorganisiert werden. Der Zeiger wird in der selben Weise wie der Expansionszeiger initialisiert und abgeändert.

Die Operation Anpassen wird durch das Beispiel in der Abb. 9 verdeutlicht. Zu beachten ist, daß die Gitterpartitionierung sowie der binäre Baum der 1. Achse (und damit auch die Verteilung der ersten Schlüsselkomponente) gegenüber dem Beispiel in Abb. 7 nicht abgeändert wurden. Trotzdem hat sich die gesamte Verteilung der Daten wesentlich verändert. So liegen in der Scheibe S(3,2) 13 Datensätze, in den Scheiben S(1,2) und S(2,2) gemeinsam nur 7 Datensätze. Da die Split- und die Mischachse der ersten Achse entspricht, fordert die Kontrollfunktion (Acc 0.25) ein Anpassen dieser Scheiben. Zunächst wird der binäre Baum an die neue Gitterpartitionierung angepaßt und dann der erste Schritt der Anpassung vollzogen, siehe Abb. 9. Die weiteren Schritte werden entsprechend zu den Schritten beim Splitten und beim Verschmelzen ausgeführt.

2.3.4 Koordination der internen Operationen

Für die internen Operationen Splitten einer Scheibe, Verschmelzen zweier Scheiben oder Anpassen dreier Scheiben muß folgende Regel eingehalten werden: Es ist nicht erlaubt mehrere dieser Operationen gleichzeitig auszuführen. Dadurch wird unsere Strategie beibehalten, nämlich eine Einfüge- oder Entferneoperation nicht durch eine Reorganisation der Datei übermäßig zu belasten. Zudem wird die Anpassung des Gitters an die Verteilung oder die dynamische Reorganisation der Datei kaum behindert.

Die oben vorgestellten Kontrollfunktionen sind nur eine Möglichkeit, das dynamische Verhalten von PLOP-Hashing auf der Basis der schrittweisen linearen Reorganisation zu steuern. Andere Kontrollfunktionen werden in dem Abschnitt 2.5 nochmals ausführlich diskutiert. Im folgenden wollen wir aus Gründen der Verständlichkeit zum einen die Kontrollfunktionen (Ex 1) und (Co 0.45) weiter beibehalten und zum anderen die interne Operation Anpassen nicht mehr betrachten. Wir möchten hier nochmals bemerken, daß auf diese Operation des PLOP-Hashings bei realen Anwendungen nicht verzichtet werden kann.

2.4 Die Suchalgorithmen

In diesem Abschnitt betrachten wir ausführlich die von PLOP-Hashing verwendeten Algorithmen für die exakte Suche und für die Bereichsanfrage. Wir geben dabei die wichtigsten Teile der Algorithmen, je nach Verständnis, in Modula-2 [Wir 85], bzw. in Umgangssprache an. Die Schwierigkeit bei der Beschreibung der Algorithmen ist die Behandlung von Ausnahmen, die durch die dynamische Organisation und die schrittweise Ausführung der internen Operationen entstehen. Deshalb setzen wir aus Gründen der Einfachheit und Verständlichkeit im folgenden voraus, daß sowohl ein Verkleinern der Datei als auch ein Anpassen eines Gitters an die Verteilung nicht vorgenommen wird. Wir berücksichtigen also nur das Erweitern der Datei. Die nachfolgend aufgeführten Bezeichnungen sind aus den früheren Abschnitten übernommen.

B_j	binärer Baum der j-ten Achse, $1 \leq j \leq d$
d	Dimension des Datenraums D
DLA	dynamische lexikographische Adreßfunktion
D=$(D_1, \ldots, D_d)$	Datenraum
$(ep_1, \ldots, ep_d)$	Expansionszeiger
$m_j = \mid P_j \mid$	Anzahl der Partitionierungspunkte in der j-ten Achse
P_j	Menge der Partitionierungspunkte
s	Splitachse

Desweiteren benutzen wir die Struktur der binären Bäume wie sie in Abb. 4 (siehe S. 19) vorgestellt wurde. Zunächst betrachten wir den Algorithmus **PLOP_EMQ**, der eine exakte Anfrage beantwortet.

```
Algorithmus PLOP_EMQ($(K_1, \ldots, K_d)$: D);
gegeben: eine durch PLOP-Hashing organisierte Datei, Schlüssel $(K_1, \ldots, K_d) \in D$
1. (* Bestimme das Indexfeld $\vec{i}$ *)
   FOR j := 1 TO d DO
       leaf := Traverse($B_j$, $K_j$);
       $i_j := leaf \uparrow .index$
   END;
2. (* Gegebenenfalls muß der Index $i_s$ neu berechnet werden *)
   IF (eine Scheibe wird gerade gesplittet) AND
       ($i_s = ep_s$) AND
       ( $(i_1, \ldots, i_{s-1}, i_{s+1}, \ldots, i_d) <^L (ep_1, \ldots, ep_{s-1}, ep_{s+1}, \ldots, ep_d)$ ) AND
       ($K_s \geq$ (dem neu eingeführten Partitionierungspunkt AB))
   THEN
       $i_s := m_s$      $(* \ m_s = \mid P_s \mid, \ AB \in P_s \ *)$
   END;
3. adr := DLA($\vec{i}, \vec{m}$);
4. Durchsuche die Kette mit Adresse adr und gebe eine entsprechende Meldung,
   ob der Datensatz gefunden oder nicht gefunden wurde.

END PLOP_EMQ;
```

Sei eine Datei mit d=2, $D = [0,1)^2$, s = 1 und ep = (1,1) gegeben. Die entsprechende Situation ist in Abb. 10 veranschaulicht.

1. Betrachten wir zunächst eine exakte Anfrage für den Schlüssel K = (0.2,0.7). In dem 1. Schritt erhalten wir das Indexfeld $\vec{i} = (0,1)$. Da nun $ep_1 \neq i_1$, gehört die Kette nicht in die Scheibe, die gerade gesplittet wird. Somit erhalten wir die Adresse DLA((0,1),(3,1)) = 2.

2. Betrachten wir eine exakte Anfrage für den Schlüssel K' = (0.7,0.2), so erhalten wir das Indexfeld $\vec{i'} = (1,0)$. Da nun $ep_1 = 1$, liegt der Datensatz in der Scheibe, die gerade erweitert wird. Weiterhin ist auch $i'_2 = 0 < 1 = ep_2$ und die Schlüsselkomponente K_2 rechts des neuen Partitionierungspunkts, so daß $i'_2 = 3$ gesetzt wird. Die Adresse ergibt sich somit aus DLA((3,0)(3,1)) = 6

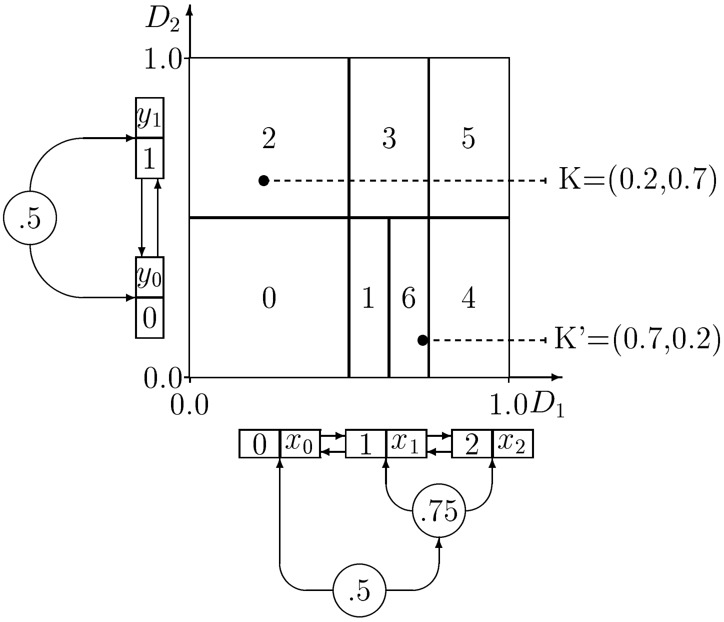


Abbildung 10: Beispiele für eine exakte Suche

Um Bereichsanfragen zu beantworten, nutzen wir die Verzeigerung auf Blattebene der binären Bäume aus. Wie für den Algorithmus **PLOP_EMQ**, berücksichtigen wir auch für die Bereichsanfrage nur den Sonderfall des Erweiterns der Datei.

Algorithmus PLOP_RQ(U,L: D);
gegeben: L=$(L_1, \dots, L_d)$, U=$(U_1, \dots, U_d)$ $\in$ D mit $L_j \leq U_j$, $1 \leq j \leq d$, sowie eine durch PLOP-Hashing organisierte Datei

```
1. FOR j := 1 TO d DO
       $\text{LeafLow}_j$ := Traverse($B_j$, $L_j$);
       $\text{LeafUp}_j$ := Traverse($B_j$, $U_j$);
   END;
   IF (eine Erweiterung wird gerade ausgeführt) THEN
       assoc := Assoc($\text{LeafUp}_s$,$\text{LeafLow}_s$);
   END;
   LeafAct := LeafLow;
2. REPEAT
       FOR j := 1 TO d DO
           $i_j$ := LeafAct↑.index
       END;
       IF ($i_s = ep_s$) AND ( $\vec{i} <^L \vec{ep}$) THEN
           IF assoc = both THEN     $(*m_s =| P_s |, AB \in P_s*)$
               adr := DLA($(i_1, \dots, i_{s-1}, m_s, i_{s+1}, \dots, i_d)$, $\vec{m}$);
               Durchsuche Kette adr nach Antworten
           ELSIF assoc = right THEN
               $i_s := m_s$
```

```
      END;
    END;
    adr := DLA($\vec{i}, \vec{m}$);
    Durchsuche Kette adr nach Antworten;
    LeafAct := NextLeaf(LeafAct, LeafUp, LeafLow);
  UNTIL LeafAct = LeafLow;
END PLOP_RQ;
```

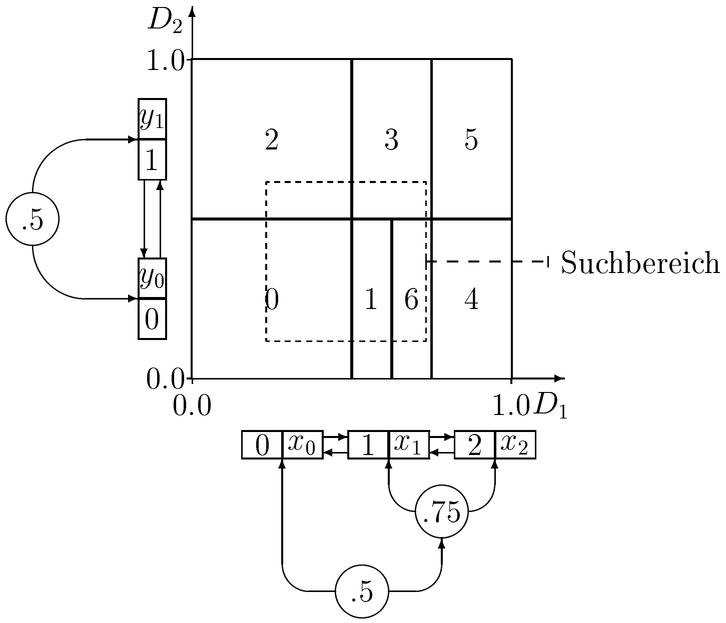


Abbildung 11: Beispiel für eine Bereichsanfrage

Der Algorithmus **PLOP_RQ** verwendet zwei Prozeduren, deren Wirkungsweise wir hier nur kurz vorstellen wollen. Falls zur Zeit einer Bereichsanfrage eine Scheibe gesplittet wird, entscheidet die Prozedur **Assoc**, welche der neu erzeugten Scheiben den Anfragebereich schneiden. Abhängig davon, ob die linke, rechte oder beide neuen Scheiben im Anfragebereich liegen, liefert die Prozedur **Assoc** als Resultat **left**, **right** oder **both** zurück. Dieser Wert wird im weiteren nur dann verwendet, falls auch Seiten aus diesen Scheiben, die bereits während dieser stückweisen linearen Erweiterung neu erzeugt wurden ($\vec{i} <^L \vec{ep}$), im Suchbereich liegen.

Die Prozedur **NextLeaf** erzeugt den nächst höheren Wert der Variablen **LeafAct** bzgl. der lexikographischen Ordnung. Durch die Verkettung der Blätter der binären Bäume ist explizit eine Ordnungsrelation gegeben. Auf dieser Ordnungsrelation basiert die lexikographische Ordnung, die von der Prozedur **NextLeaf** verwendet wird. Dabei ist der Bereich für die lexikographische Ordnung durch die Variablen **LeafUp** und **LeafLow** vorgegeben, die für jede Achse auf die am weitesten rechts, bzw. links liegende Scheibe verweisen. Ist nun **LeafAct** = **LeafUp**, d. h. es gibt keinen Nachfolger mehr, so wird **LeafLow** der Variablen **LeafAct** zugewiesen.

Um die Vorgehensweise des Algorithmus **PLOP_RQ** zu verdeutlichen, beantworten wir für die Datei, die wir bereits für den Algorithmus **PLOP_EMQ** betrachtet haben, eine

Bereichsanfrage mit L=(0.2,0.2) und U=(0.7,0.7), siehe Abb. 11. In dem ersten Schritt werden mittels der Prozedur **Traverse** folgende Blätter angesprochen: **LeafUp** = (1,1) und **LeafLow** = (0,0). Wir haben hierbei die entsprechenden Indizes der zugehörigen Blättern in den Variablen **LeafLow** und **LeafUp** notiert. Da gerade ein Erweitern der Scheibe S(1,1) ausgeführt wird, wird die Funktion **Assoc** berechnet. Der rechte Rand des Suchbereiches liegt links des neuen Partitionierungspunkts und somit liefert **Assoc** als Ergebnis **both**. Im zweiten Schritt ist nun zunächst $\vec{i} = \vec{0}$. Da $i_1 \neq 1$, ist die Bedingung des ersten IF-Statements nicht erfüllt. Somit durchsuchen wir die Kette 0 = DLA((0,0),(3,1)) nach entsprechenden Antworten. Danach wird **LeafAct** mittels der Prozedur **NextLeaf** neu angepaßt. Wir erhalten somit $\vec{i}$ = (0,1). Erneut ist die Bedingung des ersten IF-Statements nicht erfüllt und wir erhalten als nächste Adresse 2 = DLA((0,1),(3,1)). Im nächsten Durchgang der REPEAT-Schleife ist $\vec{i}$ = (1,0). Da nun (1,0) $<^L$ (1,1) = $\vec{ep}$, ist die Bedingung des IF-Statements erfüllt. Somit müssen wir zunächst die Kette mit Adresse 6 = DLA((3,0),(3,1)) und später die Kette mit Adresse 1 = DLA((1,0),(3,1)) nach Antworten durchsuchen. Im letzten Durchlauf der REPEAT-Schleife durchsuchen wir schließlich die Kette mit Adresse 3 = DLA((1,1),(3,1)). Danach ist die Bereichsanfrage abgeschlossen.

2.5 Erweiterungen und Varianten von PLOP-Hashing

Unter PLOP-Hashing kann eine ganze Familie von mehrdimensionalen dynamischen Hashverfahren verstanden werden. Durch geeignetes Setzen von Parametern, wie beispielsweise einer grundsätzlich andereren Überlauforganisation, können die Eigenschaften von PLOP-Hashing, so wie wir es in den letzten Abschnitten beschrieben haben, wesentlich verändert werden. Allen Varianten von PLOP-Hashing liegt aber dieselbe Adreßfunktion DLA zu Grunde. In den folgenden Abschnitten werden wir kurz einige Varianten von PLOP-Hashing vorstellen.

2.5.1 Partielle Erweiterungen

Das Ziel einer jeden PZS ist es insbesondere, die Datensätze gleichmäßig auf die Datenseiten zu verteilen. Setzen wir voraus, daß PLOP-Hashing eine Datei mit genau 2^L Ketten organisiert und daß die Ketten gleichmäßig mit Datensätzen gefüllt sind. Durch weitere Einfügeoperationen wird ein Splitten einer Scheibe gefordert. Nachdem dieser Split vollständig ausgeführt wurde, liegen in einer bereits gesplitteten Kette im Durchschnitt nur halb so viel Datensätze als in einer anderen (noch nicht gesplitteten) Kette. Wir erhalten somit eine ungleichmäßige Verteilung der Datensätze auf die Ketten, was sich direkt auf die Suchkosten auswirkt. Dasselbe Problem haben wir bereits für lineares Hashing angesprochen. Um die Abhängigkeit der Suchkosten von dem Verhältnis der gesplitteten zu den noch nicht gesplitteten Ketten zu reduzieren, wurde in [Lar 80] das Konzept der partiellen Erweiterungen für lineares Hashing vorgestellt. Die Verallgemeinerung von partiellen Erweiterungen für mehrdimensionales lineares Hashing wurde in [KS 86] vorgeschlagen.

Da PLOP-Hashing im Prinzip eine zu [KS 86] ähnliche Adreßfunktion verwendet, läßt sich das Konzept der partiellen Erweiterungen einfach übertragen. Wir stellen hier nur das Erweitern einer Datei vor, die durch PLOP-Hashing mit partiellen Erweiterungen organisiert wird. Statt wie bisher bei einem Schritt der Erweiterung stets eine Kette in zwei Ketten aufzuspalten, betrachten wir zwei benachbarte Ketten, deren Datensätze auf drei Ketten gleichmäßig aufgeteilt werden. Dabei entsprechen die Adressen von zwei dieser drei Ketten genau den Adressen der ursprünglichen Ketten, während die Primärseite der dritten Kette neu am Ende der Primärdatei angehängt wird. Entsprechend zu diesem Erweitern von nebeneinander liegenden Ketten, benötigen wir eine Kontrollfunktion, die ein Erweitern von nebeneinander liegenden Scheiben auslöst. Wir könnten z. B. folgende Kontrollfunktion verwenden:

> Bestimme nach einem Einfügen eines Datensatzes in der aktuellen Splitachse s das Paar von benachbarten Scheiben ($S(i_s, s)$, $S(j_s, s)$), $i_s \neq j_s$, $0 \leq i_s, j_s < m_s$, welches die maximale Anzahl von Datensätzen enthält. Falls $(l(i_s, s)+l(j_s, s))/2.0 > 100\%$ gilt, so wird eine Erweiterung der beiden Scheiben gefordert.

Wie oben beschrieben wird das Erweitern der zwei Scheiben linear ausgeführt. Das Prinzip partieller Erweiterungen kann insofern verallgemeinert werden, daß wir beim Erweitern der Datei c Ketten bzw. Scheiben, $c \geq 1$, betrachten, deren Datensätze auf (c + a) Ketten bzw. Scheiben, $a \geq 1$, gleichmäßig aufgeteilt werden. Für $a = 1$ und $c = 1$ erhalten wir z. B. das ursprüngliche PLOP-Hashing. Um die Kosten in einem Erweiterungsschritt niedrig zu halten, wurde in [Lar 80] empfohlen $c \leq 3$ und $a = 1$ zu wählen. Durch obige Kontrollfunktion ergibt sich auch eine höhere Span, was in Kombination mit der gleichmäßigeren Belegung der Datenseiten die Kosten insbesondere von Bereichsanfragen reduziert.

Die Technik der partiellen Erweiterungen läßt sich, invers zum Erweitern, auf das Verkleinern der Datei anwenden. Beispielsweise können Datensätze aus drei benachbarten Ketten bzw. Scheiben auf zwei Ketten bzw. Scheiben umverteilt werden.

2.5.2 Kontrollfunktionen

Unter einer Kontrollfunktion verstehen wir eine Bedingung, die nach gewissen Ereignissen überprüft wird. Falls die Bedingung erfüllt ist, wird eine der internen Operationen, wie Splitten, Verschmelzen und Anpassen, ausgeführt. PLOP-Hashing stellt keine Bedingung daran, wie solch eine Regel auszusehen hat. Es besteht also die Möglichkeit, verschiedene Kontrollfunktionen dem Benutzer anzubieten, der sich für seine Anwendung dann die geeigneteste Kontrollfunktion auswählen kann. Wir stellen deshalb ergänzend zu den bereits betrachteten weitere Kontrollfunktionen vor, wobei wir uns auf diejenigen beschränken, die ein Erweitern und Verkleinern der Datei auslösen.

Zuerst betrachten wir für zwei vorgegebenen Schranken S_L, S_U mit $0 < S_L < S_U < 1$ folgendes Paar von Kontrollfunktionen:

(Ex,Span,S_U) Falls die Span der Datei die Schranke S_U überschreitet, so bestimme in der Splitachse s die Scheibe mit der maximalen Anzahl von Datensätzen und beginne mit dem Erweitern dieser Scheibe.

(Co,Span,S_L) Falls die Span der Datei unter die Schranke S_L absinkt, so bestimme in der Mischachse das Paar von Scheiben mit der minimalen Anzahl von Datensätzen und beginne mit dem Verschmelzen der zugehörigen Ketten.

Diese Kontrollfunktionen verallgemeinern die für lineares Hashing empfohlenen Kontrollfunktionen [Lit 80], wobei die Möglichkeit der stückweisen linearen Erweiterung der Datei ausgenutzt wird. Sie garantieren, daß die Span einer Datei zwischen den Schranken S_L und S_U liegt. Der prinzipielle Unterschied zu den bisher betrachteten Kontrollfunktionen für PLOP-Hashing ist, daß ein globales Ereignis das Erweitern einer Scheibe oder Verschmelzen mehrerer Scheiben auslöst. Im Gegensatz dazu haben die Kontrollfunktionen (Ex α) und (Co β) ein Erweitern bzw. Verkleinern der Datei durch ein lokales Ereignis (nämlich der Belegung einer Scheibe) ausgelöst. Die "lokalen" Kontrollfunktionen sorgen für eine sehr gute Anpassung der Gitterpartitionierung an die Verteilung der Daten. Andererseits ergeben sich hohe Schwankungen für die Span und die Kosten einer exakten Suche. Die Leistung hängt dabei davon ab, ob viele Scheiben kurz hintereinander erweitert wurden oder über einen längeren Zeitraum hinweg keine Scheibe erweitert wurde. Für die oben eingeführten "globalen" Kontrollfunktionen gilt, daß sie insbesondere für eine permanent größer bzw. kleiner werdende Datei eine sehr gleichmäßige Leistung des PLOP-Hashings garantiert. Der Nachteil dieser Kontrollfunktionen ist, daß die Belegung der Scheiben sehr unterschiedlich sein kann und somit mit einer nicht so gleichmäßigen Belegung der Ketten zu rechnen ist.

Eine weitere Möglichkeit für eine Kontrollfunktion besteht darin, die Längen der Ketten zu begrenzen. Sei $cl \geq 1$ eine ganzzahlige Zahl, so betrachten wir folgende Kontrollfunktionen:

(Ex,Kl,cl) Bestimme in der Splitachse s die Scheibe, die die längste Kette von Datenseiten enthält. Falls die maximale Anzahl von Seiten pro Kette cl übersteigt, so beginne diese Scheibe zu erweitern.

(Co,Kl,cl) Bestimme in der Mischachse das Paar von Scheiben, deren Summe der Datensätze aus den jeweils am stärksten belegten Ketten minimal ist. Falls diese Summe kleiner als $\lceil cl/2 * b \rceil$ ist, so beginne mit dem Verschmelzen dieser Scheiben.

Unter der Voraussetzung alle Ketten einer Scheibe in einem Schritt zu erweitern, garantiert die Kontrollfunktion (Ex,Kl,cl), daß alle Ketten aus nicht mehr als cl Seiten bestehen. Man beachte, daß die durch ein Verschmelzen gewonnene Scheibe i. a. nicht sofort wieder gesplittet wird. Für die Realisierung dieser Kontrollfunktionen müssen die Blätter der binären Bäume entsprechend angepaßt und erweitert werden.

Anhand dieser Kontrollfunktionen läßt sich die Ähnlichkeit der Gitterdatei und PLOP-Hashing aufzeigen. Wählen wir nämlich cl=2, so garantiert uns diese Variante des PLOP-Hashing eine maximale Kettenlänge von zwei Datenseiten. Darüber hinaus ergibt sich durch Reduzierung der Kapazität einer Primärseite auf null, daß PLOP-Hashing statt einer Primärdatei eine Adreßtabelle organisiert. Der einzig prinzipielle Unterschied zu der Gitterdatei besteht darin, daß in der Adreßtabelle der Gitterdatei mehrere Zeiger auf die gleiche Datenseite verweisen können. Wie im nächsten Abschnitt gezeigt wird, kann durch eine geeignete Wahl der Überlauforganisation selbst dieser Unterschied beseitigt werden. Wir können somit durch eine Implementierung sowohl die Gitterdatei als auch PLOP-Hashing realisieren, wobei durch geeignetes Setzen von Parametern das jeweilige Verfahren spezifiziert wird.

2.5.3 Organisation der Überlaufsätze

Ein Datensatz wird als Überlaufsatz oder Überläufer bezeichnet, wenn dieser in der zugehörigen Seite der Primärdatei nicht eingefügt werden kann und deshalb gesondert behandelt werden muß. Für PLOP-Hashing haben wir Überläufer bisher separat in Seiten einer Sekundärdatei gehalten, wobei die Überlaufseiten mit den Seiten der Primärdatei verkettet wurden (Seitenverkettung). Da PLOP-Hashing das Prinzip von linearem Hashing verallgemeinert, können auch speziell für lineares Hashing vorgeschlagene Überlauforganisationen, wie z. B. die rekursive Dateiorganisation [RS 84] oder Überlauforganisation in der Primärdatei [Lar 85], für PLOP-Hashing genutzt werden.

Für PLOP-Hashing ist eine rekursive Überlauforganisation nur mit hohem Aufwand zu realisieren. Unter rekursiv verstehen wir hierbei, daß Überlaufsätze der Primärdatei (Datei der Stufe 1) in einer Datei der Stufe 2 abgelegt werden, die wiederum durch PLOP-Hashing organisiert wird. Alle Überläufer dieser Datei werden in einer separaten Datei der Stufe 3 gehalten, usw.. Das Problem bei PLOP-Hashing ergibt sich hierbei durch die an die Verteilung angepaßte Partitionierung des Datenraums, sowie durch die freie Wahl der Scheibe in welcher erweitert wird. Im Gegensatz zum rekursivem linearen Hashing ist nicht mehr garantiert daß eine Seitenregion der Datei auf Stufe i in genau einer Seitenregion der Datei auf Stufe (i+1) liegt, $i \geq 1$. Damit müssen nach dem Splitten einer Seite i. a. Bereichsanfragen an die höherstufigen Dateien gestellt und entsprechend Datensätze aus diesen Dateien in die neu erzeugten Seiten umgespeichert werden. Die Frage stellt sich, ob dann das Verfahren überhaupt noch als dynamisch bezeichnet werden kann.

Im folgenden stellen wir eine weitere Technik zur Organisation von Überläufern vor, die wir als gepackte Seitenverkettung bezeichnen. Das Problem bei der üblichen Seitenverkettung ist die niedrige Belegung vieler Sekundärseiten. Offensichtlich ergibt sich dadurch eine geringe Span. Eine Möglichkeit die Span zu erhöhen ist, die Größe einer Sekundärseite künstlich zu verkleinern. Dadurch werden aber gleichzeitig die Suchkosten für jegliche Anfragen erhöht. Die Verbesserung der Span in der Sekundärdatei kann, unter der Gewährleistung konstanter Suchkosten, durch Zuordnung verschiedener Primärseiten zu einer Sekundärseite erreicht werden. Falls zwei Sekundärseiten jeweils unter 50% (oder unter einer vorgegebenen Schranke) gefüllt sind, werden diese miteinander verschmolzen und die Zeiger in den zugehörigen Primärseiten neu angepaßt. Bei gepackter Seitenverkettung erhöhen sich somit die Kosten beim Einfügen und Löschen von Datensätzen. Weiterhin sollten, falls es zu einer Primärseite $r * b + s$, $r, s \geq 1$, Überlaufsätze gibt, dieser Seite genau r Überlaufseiten exklusiv zugeordnet sein. Durch diese einfache Organisation können wir eine Span von 50 % in der Sekundärdatei garantieren, wobei der Aufwand für eine exakte Suche dem der üblichen Überlauforganisation der Seitenverkettung entspricht. Für eine Bereichsanfrage benötigen wir genau dann weniger Zugriffe, falls "gepackte" Sekundärseiten betroffen sind, wobei die Bereiche von mindestens zwei zugehörigen Ketten den Anfragebereich schneiden.

Insbesondere zur Unterstützung von Bereichsanfragen ist das Packen von Sekundärseiten, deren Datensätze nah beieinander liegen, vorteilhaft. Eine Realisierung dieses Konzepts könnte genau dem der Gitterdatei entsprechen, das in [NHS 84] als Buddy-System bezeichnet wird. Das Buddy-System regelt, wann Zeiger in der Adreßtabelle auf die selbe Datenseite verweisen dürfen.

Durch die Überlauforganisation der Seitenverkettung in Kombination mit dem Buddy-

System und unter Verwendung der Kontrollfunktionen (Ex,KL,2) und (Co,KL,2), entsteht aus PLOP-Hashing die Gitterdatei, wie sie in [NHS 84] vorgestellt wurde. Die Analogie zwischen PLOP-Hashing und der Gitterdatei dient dazu, aufzuzeigen, daß unter PLOP-Hashing eine ganze Familie von Zugriffsstrukturen zu verstehen ist. Durch einfaches Setzen von Parametern kann der Benutzer sich die für seine Bedürfnisse geeigneteste Variante zusammenstellen.

2.5.4 Weitere Eigenschaften des PLOP-Hashings

Zum Abschluß unserer Betrachtungen stellen wir weitere wichtige Eigenschaften des PLOP-Hashings vor. Bisher sind wir aus Gründen einer verständlichen Beschreibung des Verfahrens auf diese praktisch interessanten Eigenschaften nicht näher eingegangen.

Bisher haben wir bei PLOP-Hashing stets angenommen, daß der Datenraum bei Initialisierung der Datei fest vorgegeben werden muß und während der Existenz dieser Datei nicht veränderbar ist. Diese Annahme wird für PLOP-Hashing nicht benötigt. Im Gegensatz dazu setzen Verfahren wie die Gitterdatei [NHS 84] oder nahezu alle auf Z-Ordnung basierenden PZS, wie das Interpolationsverfahren [Bur 83] oder das BANG-File [Fre 87], einen *festen* Datenraum voraus. Da bei Initialisierung, beispielsweise der Gitterdatei, der Benutzer den Datenraum nicht exakt spezifizieren kann, wird dieser oft stark überschätzt. Dadurch muß die Gitterdatei z. T. leere Datenbereiche partitionieren, was deren Effizienz negativ beeinflußt.

Eine weitere nicht benötigte Annahme war bisher, daß während des Lebenszykluses einer Datei die Dimension der Schlüssel $K = (K_1, \ldots, K_d)$ unveränderbar ist. PLOP-Hashing erlaubt es, einen Teil des Informationsteils der Datensätze zusätzlich als Schlüsselkomponente zu deklarieren. Ohne dabei die Datei reorganisieren zu müssen, kann PLOP-Hashing die Daten nach diesen neu deklarierten Schlüsselkomponenten organisieren. Betrachten wir unseren Schlüssel nicht als Identifikationsschlüssel sondern nur als Sortierschlüssel eines Datensatzes (d. h. Duplikate von Schlüsseln sind erlaubt), so kann insbesondere der Fall auftreten, daß der Datenraum durch die Datensätze in der Datei abgedeckt ist. Dadurch wäre ein Splitten von Datenseiten bzgl. der bisherigen Schlüsselkomponenten nicht sinnvoll. Werden aber Datenseiten nicht gesplittet, so werden Datensätze nur noch als Überlaufsätze organisiert. Durch eine Erweiterung des Schlüssels um eine weitere Dimension könnte dieses Problem eliminiert werden.

3 Der Buddy-Hashbaum

In den letzten Abschnitten haben wir MDH betrachtet, die den d-dimensionalen Datenraum D mit Hilfe eines d-dimensionalen orthogonalen Gitters unterteilen. Für schwach, bzw. nicht-korrelierte Daten besitzen diese Verfahren eine annähernd optimale Leistung. Im Fall stark korrelierter oder gar funktional abhängiger Daten degeneriert die Leistung dieser Verfahren. Betrachten wir z. B. Schlüssel K=$(K_1, \ldots, K_d)$ mit $K_1 = \ldots = K_d$, d. h. die Schlüssel liegen alle auf der Diagonalen des d-dimensionalen Datenraums D, so kostet eine erfolgreiche exakte Suche für PLOP-Hashing $O(n^{1-1/d})$ Zugriffe. Werden solch verteilte Daten mit der Gitterdatei organisiert, so wächst die Adreßtabelle der Gitterdatei mit $O(n^d)$ an und ein Einfügen eines Datensatzes kann $O(n^{d-1})$ Zugriffe erfordern. Offensichtlich sind solche PZS nicht zur Organisation korrelierter Daten geeignet.

In den letzten Jahren sind vermehrt Verfahren vorgestellt worden, die das worst-case Verhalten von MDH, wie z. B. der Gitterdatei, speziell für korrelierte Schlüsselkomponenten verbessern. Die meisten dieser Verfahren sind ein Kompromiß zwischen mehrdimensionalem erweiterbaren Hashing (MEH) ([Tam 82], [Oto 84]), bzw. der Gitterdatei [NHS 84], und dem k-d-B-Baum [Rob 81]. Aus diesem Grund bezeichnen wir auch diese Verfahren als Hashbäume. Im nächsten Abschnitt werden wir eine kurze Übersicht der Hashbäume geben und insbesondere die Eigenschaften der verschiedenen Verfahren herausstellen. In den weiteren Abschnitten werden wir dann den Buddy-Hashbaum vorstellen.

I

3.1 Mehrdimensionale Hashbäume

Der wesentliche Grund für die Entwicklung von Hashbäumen ist das starke Anwachsen des Directories bei der Gitterdatei oder MEH. Schon bei ersten Implementierungen dieser Verfahren ist das Directory in einem Baum der Höhe 2 realisiert worden. Das Directory wurde hierbei in ein Wurzeldirectory und mehrere Subdirectories unterteilt. Dabei wurde vorausgesetzt, daß das Wurzeldirectory stets im Hauptspeicher gehalten werden kann. Das Wurzeldirectory selbst entspricht genau der üblichen Organisation eines Directories mit der Ausnahme, daß die Zeiger nicht auf Datenseiten sondern auf Subdirectories verweisen, siehe Abb. 12. Die Organisationsform der Subdirectories entspricht dem eines üblichen Directories, wobei die Anzahl der Einträge in einem Subdirectory durch die Größe einer Seite beschränkt ist. Entsprechend zu dieser Organisation, werden diese Verfahren auch als 2-Level Gitterdatei oder als 2-Level MEH bezeichnet. Der Vorteil dieser Verfahren ist, daß sie unter der Annahme eines im Hauptspeicher liegenden Wurzeldirectories weiterhin die Eigenschaften der 1-Level-Verfahren (wie z. B. die 2-Zugriffs-Garantie bei exakten Anfragen) besitzen, wobei aber das Wachstum des gesamten Directories erheblich reduziert werden konnte. Die Größe des Directories wird maximal um den Faktor c reduziert, wobei c, $c > 0$, die Kapazität einer Directoryseite ist.

Für viele Anwendungen genügt ein 2-Level Directory den Anforderungen. Für stark korrelierte Daten ist aber mit einem schnellen Anwachsen des Wurzeldirectories zu rechnen, so daß die Annahme, es im Hauptspeicher halten zu können, in solchen Fällen nicht gerechtfertigt ist. Daraus ergibt sich die Idee, das Wurzeldirectory selbst wieder als 2-Level

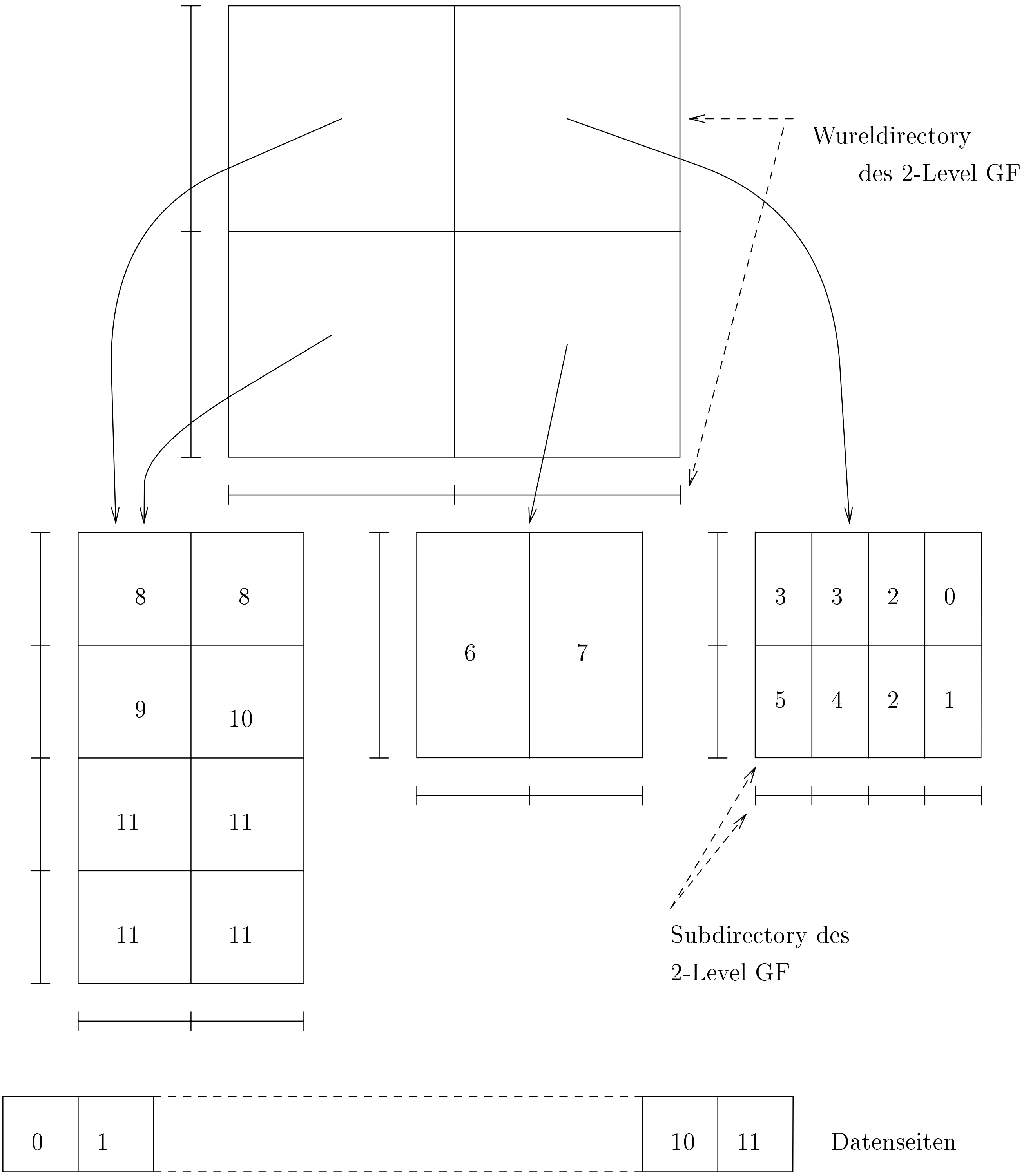


Abbildung 12: Organisation der 2-Level Gitterdatei

Directory zu organisieren. Durch Verallgemeinerung dieses Prinzips ergibt sich ein balancierter Baum. In jedem inneren Knoten, dieses sogenannten Hashbaums, liegt ein Gitterdirectory, dessen Größe durch die Größe einer Seite beschränkt ist. Verschiedene Varianten dieser Hashbäume wurden vorgestellt, wie die Interpolations-Gitterdatei [Ouk 85], der balancierte mehrdimensionale erweiterbare Hashbaum (BMEH) [Oto 86] und die Multi-Level Gitterdatei [WK 85]. Unter der Annahme, daß das Wurzeldirectory im Hauptspeicher liegt, werden nun für eine exakte Anfrage i. a. mehr als zwei Zugriffe benötigt. Die Höhe des Baums und damit die Anzahl der Zugriffe für eine exakte Suche kann im schlechtesten Fall nur von der Anzahl $w,\ w > 0$, der Bits abhängen, die zur Repräsentation eines Schlüssels benötigt werden. Im folgenden werden wir w auch als Auflösung des Datenraums D bezeichnen. Unter der Annahme $\mid D \mid = 2^w$, ist z. B. die maximale Höhe des erweiterbaren Hashbaums durch

$$\left\lceil \frac{w - \lfloor \log_2 b \rfloor}{\lfloor \log_2 c \rfloor} \right\rceil \tag{4}$$

gegeben, wobei b die Kapazität einer Datenseite und c die Kapazität einer Directoryseite ist. Setzen wir $b = 2^6$ und $c = 2^7$ voraus, so ergibt sich für $w \leq 27$ maximal eine Höhe von 3 und für $w \leq 34$ maximal eine Höhe von 4. Für relativ "kleine" Wertebereiche besitzen somit Hashbäume eine akzeptabele Leistung. Für große Wertebereiche, wie etwa w=512, kann der Baum sehr hoch werden (maximal Höhe 73).

Bisher genannte Verfahren teilen den Datenraum disjunkt in rechteckige Regionen auf, wobei genau eine Region zu einer Datenseite gehört. Das Problem bei einer solch restriktiven Anforderung für Seitenregionen ist, daß sehr viele unterfüllte Seiten existieren können. Nebeneinanderliegende Seiten können nicht miteinander verschmolzen werden, da die resultierende Region nicht die Form eines Rechtecks hat. Solch ein Verhalten wird durch zusätzliche Anforderungen verstärkt. So werden z. B. bei der Interpolations-Gitterdatei nur dann zwei Seiten miteinander verschmolzen, falls diese Seiten aus dem Splitten einer Seite entstanden sind. Insbesondere können diese Verfahren keine hohe Span garantieren. Ähnlich wie bei der Gitterdatei ist nur bei Gleichverteilung der Daten mit einer durchschnittlichen Span von 69% zu rechnen. Man beachte dabei, daß sich bei diesen Verfahren die Span auf die Leistung einer Bereichsanfrage auswirkt.

Als Vorteil rechteckiger Seitenregionen erweist sich die relativ einfache Umsetzung in Datentypen und Algorithmen bei einer Implementierung. Alle Algorithmen für Einfügen, Löschen und Suchen sind vollständig vorgegeben und weisen keine Spezialsituationen auf, die die Komplexität und Fehlerträchtigkeit eines Programms wesentlich erhöhen. Zudem erwarten wir, daß diese Verfahren wenig CPU-Zeit benötigen. Eine der am meisten ausgeführten Operation bei Hashbäumen ist das Testen, ob ein Datensatz in, bzw. außerhalb, einer Seitenregion liegt. Diese Operation kann für rechteckige Regionen schnell ausgeführt werden.

Im nächsten Abschnitt werden wir den Buddy-Hashbaum vorstellen, der gegenüber den bisher vorgestellten Hashbäumen ein verbessertes Leistungsverhalten aufweist. Eine Vorarbeit zum Buddy-Hashbaum wurde in [Fra 87] geleistet. Charakteristische Eigenschaften des Buddy-Hashbaums sind, daß zum einen die Zeiger auf Daten- und Directoryseiten paarweise verschieden sind und zum anderen leere Seitenregionen nicht im Directory repräsentiert werden. Zudem wird die Verteilung der Daten, die ein maximales Anwachsen des Hashbaums hervorrufen, praktisch nie auftreten. Schließlich werden wir eine "gepackte" Variante des Buddy-Hashbaums vorstellen, die eine Span von 50% garantiert.

Eine interessante Verallgemeinerung der Interpolations-Gitterdatei [Ouk 85] und des binären BD-Baum [OsS 83] stellt das BANG-File [Fre 87] dar. Das BANG-File garantiert (praktisch) ein lineares Wachstum des Directory in der Anzahl der Datensätze. Da der Hashbaum höhenbalanciert ist, ergibt sich somit eine logarithmische Suchzeit bei einer exakten Suche. Das Verhalten des BANG-File basiert auf der Einteilung des Datenraums in nicht notwendigerweise rechteckige Seitenregionen. Ähnlich zm BANG-File verhält sich der hB-Baum [LS 87]. Der Hauptunterschied ist, daß der hB-Baum interne Knoten nicht durch ein Hashverfahren, sondern durch einen kd-Baum organisiert. In diesem Sinne ist der hB-Baum kein Hashbaum. Das Hauptproblem des hB-Baums ist das Löschen von Datensätzen und das damit verbundene Verschmelzen von Datenseiten bzw. Directoryseiten und den zugehörigen Regionen. Es bleibt zu untersuchen, wie effizient Bereichsanfragen und partielle Anfragen durch diese Verfahren unterstützt werden. Man beachte hierzu, daß Seitenregionen des hB-Baums oder des BANG-Files nicht notwendigerweise zusammenhängend sein müssen. Eine Vielzahl nicht zusammenhängender Regionen würde insbesondere die Leistung bei komplexen Anfragen mindern.

Zum Abschluß gehen wir kurz auf wichtige Eigenschaften mehrdimensionaler Hashbäume bzw. Bäume ein, die sich positiv auf die Leistung auswirken. Dabei setzen wir voraus, daß die Verfahren ordnungsbewahrend sind und daß sich die Partitionierung des Datenraums an die jeweilige Verteilung der Daten anpaßt.

(K1) In einem Directory sollen alle Zeiger paarweise verschieden sein.

(K2) Eine exakte Suche, Einfügen und Löschen von Datensätzen sollten möglichst auf einen Pfad von der Wurzel zu einem Blatt des Baums beschränkt bleiben.

Die Eigenschaften (K1) und (K2) werden von keinem der bisherigen Verfahren gleichzeitig erfüllt. So kann das BANG-File, je nach Implementierung, entweder die Eigenschaft (K1) oder die Eigenschaft (K2) erfüllen. Für den hB-Baum ist das Verkleinern der Datei bisher überhaupt nicht berücksichtigt worden, so daß Eigenschaft (K2) nicht erfüllt ist. Die Eigenschaft (K1) wird von BMEH und der Interpolations-Gitterdatei insbesondere deshalb nicht erfüllt, da leere Regionen durch NIL-Zeiger im Directory repräsentiert werden. Die einzigen Verfahren, die die Bedingungen (K1) und (K2) gleichzeitig erfüllen sind die mehrdimensionalen B-Bäume, die, wie bereits in Abschnitt 1.2 aufgezeigt wurde, nicht ordnungsbewahrend sind.

(K3) Eine Seitenregion sollte möglichst klein sein, d. h. sie sollte möglichst nicht größer als das minimal umgebende Rechteck der Punkte sein, die in dieser Region liegen.

Durch möglichst kleine Seitenregionen garantiert uns ein Hashbaum, bei komplexen Anfragen nicht unnötig auf Seiten zuzugreifen. Alle bisher betrachteten Verfahren mißachten diese Forderung, da beim Splitten einer Seite die Seitenregion disjunkt in zwei Regionen aufgeteilt wird. Im Gegensatz dazu sollte ein Seitenbereich "minimal" in zwei Regionen aufgeteilt werden, siehe Abb.13. Man beachte hierbei, daß Hashbäume speziell für Wertebereiche mit hoher Auflösung sinnvoll sind, da relativ zu der maximal möglichen Anzahl von Datensätzen nur wenige Datensätze in der Datei abgelegt werden. Es werden also große Bereiche des Datenraums nicht belegt, die *keiner* Seitenregion zugeordnet werden sollten.

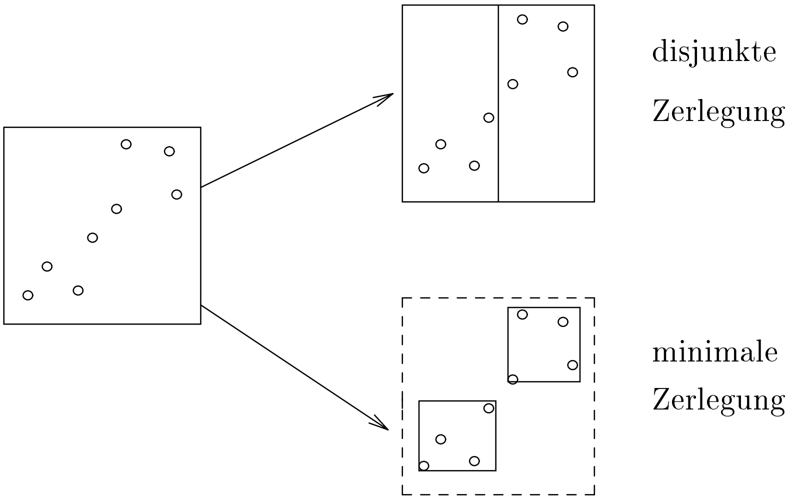


Abbildung 13: Minimales und disjunktes Zerlegen eines Seitenbereiches

(K4) Seitenregionen sollten möglichst den Anfragebereichen angepaßt sein.

Falls wir keinerlei Kenntnis über die Art und Verteilung der Anfragen besitzen, ist es am sinnvollsten alle Achsen des Datenraums in etwa gleich zu behandeln, so daß die Kosten für eine partielle Anfrage unabhängig von den spezifizierten, bzw. nicht spezifizierten, Achsen sind. Da die Anfrageregionen von partiellen Bereichsanfragen Rechtecke sind, sollten auch die Seitenregionen, wenn möglich, Rechtecke sein.

Die Anforderungen (K1) - (K4) werden von keinem bekannten Verfahren erfüllt, auch nicht vom Buddy-Hashbaum, den wir im nächsten Abschnitt genauer vorstellen werden. Der Buddy-Hashbaum kommt aber sehr nahe an diese Anforderungen heran.

3.2 Datenstruktur und Adreßfunktion

Im folgenden nehmen wir an, daß zu vorgegebenen ganzzahligen Werten z_j, $z_j > 0$, $1 \leq j \leq d$, der d-dimensionale Datenraum D=$(D_1, \ldots, D_d)$ wie folgt gegeben ist:

$$D_j = \{ \sum_{i=1}^{z_j} b_i^j * 2^{-i} \;\mid\; b_i^j \in \{0,1\},\ 1 \leq i \leq z_j \},\ \ 1 \leq j \leq d$$

Hierbei bezeichnen wir z_j als Auflösung des j-ten Wertebereichs D_j, $1 \leq j \leq d$, und $w = \sum_{j=1}^{d} z_j$ als Auflösung des Datenraums D. Der Einfachheit halber setzen wir für alle Wertebereiche D_j, $1 \leq j \leq d$, dieselbe Auflösung z, $z > 0$, voraus, so daß $w = d * z$ gilt. Wir betrachten also einen diskreten Datenraum D, in welchem genau 2^w, $w = d * z$, $z > 0$, Schlüsselwerte liegen. Man beachte, daß das kleinste von null verschiedene Element im Wertebereich D_j durch 2^{-z} gegeben ist. Als Abschluß der Menge D bezeichnen wir die Menge $\overline{D} = (\overline{D}_1, \ldots, \overline{D}_d)$, wobei $\overline{D}_j := D_j \cup \{1.0\}$, $1 \leq j \leq d$.

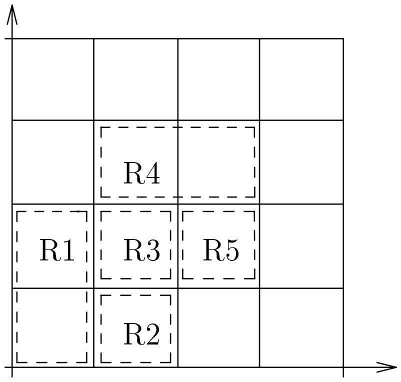


Abbildung 14: Veranschaulichung von B-Rechtecken

Definition 1
Sei $w = z * d, z > 0$, die Auflösung des Datenraums D und seien $a, b \in \overline{D}$ mit $a = (a_1, \ldots, a_d)$, $b = (b_1, \ldots, b_d)$ und $a_j < b_j, 1 \leq j \leq d$.

1. *Unter einem Rechteck $R := R(a, b) \subseteq D$ des Datenraums D verstehen wir die Teilmenge*

$$R(a, b) := \{x \in D \mid a_j \leq x_j < b_j, 1 \leq j \leq d\}$$

2. *Ein Rechteck $R(a, b) \subseteq D$ erfüllt die B-Eigenschaft, falls es zu jedem $j \in \{1, \ldots, d\}$ ein $i_j \in \{0, \ldots, z\}$ gibt, so daß folgende drei Bedingungen erfüllt sind:*
   - *(a)* $\lfloor a_j * 2^{i_j} \rfloor = \lfloor (b_j - 2^{-z}) * 2^{i_j} \rfloor$
   - *(b)* $\lfloor a_j * 2^z \rfloor \; DIV \; 2^{i_j} = 0$
   - *(c)* $\lfloor (b_j - 2^{-z}) * 2^z \rfloor \; DIV \; 2^{i_j} = 2^{z - i_j} - 1$

   Wir bezeichnen solche Rechtecke kurz auch als B-Rechtecke.

3. *Sei eine Menge von disjunkten B-Rechtecken $\mathcal{R} = \{R_1, \ldots, R_m\}, m > 1$, mit $R_j \subseteq D, 1 \leq j \leq m$, gegeben. Zwei B-Rechtecke $R, S \in \mathcal{R}$ sind Buddies zueinander, falls*
   - *(a)* $R \cap S = \emptyset$
   - *(b) es gibt ein B-Rechteck $T \notin \mathcal{R}$ mit $T \supseteq R \cup S$ und $T \cap U = \emptyset, U \in \mathcal{R} \setminus \{R, S\}$*

Bemerkungen:

1. Ein Rechteck des Datenraums D ist genau dann ein B-Rechteck, falls es durch sukzessives Halbieren des Datenraums entstanden ist. Dabei spielt die Reihenfolge der Achsen keine Rolle, bezüglich der die Halbierung des Datenraums vorgenommen wird. In unserem Beispiel auf der linken Seite in Abb. 14 sind die Rechtecke R_1, R_2, R_3 und R_5 B-Rechtecke, während das Rechteck R_4 kein B-Rechteck ist. Das B-Rechteck R_1 ist kein Buddy zu R_2 oder R_3, R_3 ist kein Buddy zu R_5, aber R_2 ist ein Buddy zu R_3.

2. Die formale Definition eines B-Rechtecks läßt sich umgangssprachlich wie folgt wiedergeben. Für jedes Intervall, das wir durch Projektion eines B-Rechtecks auf eine Achse erhalten, gilt:

- die ersten Bits vom linken und rechten Eckpunkt sind identisch
- alle anderen Bits des linken Eckpunktes sind null
- alle anderen Bits des rechten Eckpunktes sind eins

3. Die maximale Anzahl von Buddies zu einem B-Rechteck ist mindestens d. Im Unterschied zur Gitterdatei können Situationen auftreten, in denen es B-Rechtecke mit mehr als d Buddies gibt. So besitzt z. B. das B-Rechteck S in Abb. 14 (rechte Seite) drei Buddies. In dem Abschnitt 3.4 werden wir eine untere Schranke für die maximale Anzahl von Buddies herleiten.

Die Idee des Buddy-Hashbaums ist es, den Datenraum in rechteckige Seitenregionen aufzuteilen, die die B-Eigenschaft erfüllen. Im Gegensatz dazu stellen BMEH und die Interpolations-Gitterdatei zusätzliche Forderungen an ihre Seitenregionen, so daß sie nicht die Flexibilität des Buddy-Hashbaums besitzen. Bereits für die Gitterdatei wurde vorgeschlagen, den Datenraum *vollständig* in B-Rechtecke zu unterteilen. Die B-Eigenschaft der Seitenregionen sorgt sowohl bei der Gitterdatei als auch beim Buddy-Hashbaum für ein dynamisches Verhalten der Strukturen. Läßt man diese Forderung fallen, so kann insbesondere das Löschen von Datensätzen leicht zu Verklemmungen und zu einer niedrigen Span führen.

Bevor wir auf die Organisation des Directory eingehen, betrachten wir ausführlich ein Beispiel, das in Abb. 15 illustriert ist. Wir setzen hierbei voraus, daß maximal 4 Datensätze in einer Datenseite und 5 Directoryeinträge in einer Directoryseite untergebracht werden können. In Abb. 15 a haben wir eine Situation, in der 5 Datensätze in einer Datenseite abgelegt werden sollen und deshalb ein Splitten dieser Seite verlangt wird. Dies wird durch Einfügen einer den Datenraum halbierenden Partitionierungslinie senkrecht zur ersten Achse realisiert. Es entsteht hierbei eine neue Datenseite. Nach weiteren Einfügungen wird wiederum ein Splitten der linken Datenseite verlangt, siehe Abb. 15 b. Die zugehörige Seitenregion wird nun bzgl. der zweiten Achse halbiert. Da wir nur die Seitenregionen halbieren, kann der Buddy-Hashbaum nicht garantieren, daß in beiden Seiten in etwa gleich viele Datensätze liegen. Nach weiteren Einfügungen läuft die linke untere Seite über (siehe Abb. 15 c). Zunächst wird versucht, die zugehörige Region bzgl. der ersten Achse zu splitten. Da in der linken Hälfte kein Datensatz liegt, wird die rechte Hälfte nochmals bzgl. der zweiten Achse halbiert. Dadurch erhalten wir zwei nicht leere Seitenregionen. Der rechteckige Bereich, in dem keine Datensätze vorkommen, und der durch das Splitten in Abb. 15 c entstanden ist, wird in dem Directory nicht mehr repräsentiert. Diese Eigenschaft unterscheidet den Buddy-Hashbaum von bisherigen Verfahren. Durch weitere Einfügungen kann es zu einem Überlauf der rechten Datenseite kommen. Wiederum muß ein neuer Directoryeintrag in die Directoryseite eingefügt werden, siehe Abb. 15 d. Schließlich ergibt sich in Abb. 15 e durch einen Split der oberen rechten Seite ein Überlauf der Directoryseite. Um die Einträge der Directoryseite auf zwei Seiten neu aufzuteilen, nehmen wir eine Partitionierungslinie, die den Datenraum halbiert und keine bisherige Seitenregion schneidet. Wir speichern die so erzeugten B-Rechtecke in der Wurzel unseres Directories, wobei Zeiger auf die entsprechenden neuen Directoryseiten verweisen, siehe Abb. 15 e.

Es ergibt sich die Frage nach der Organisationsform des Directories und der Directoryseiten. Eine einfache Methode wäre, das einem Zeiger zugeordnete Rechteck direkt durch

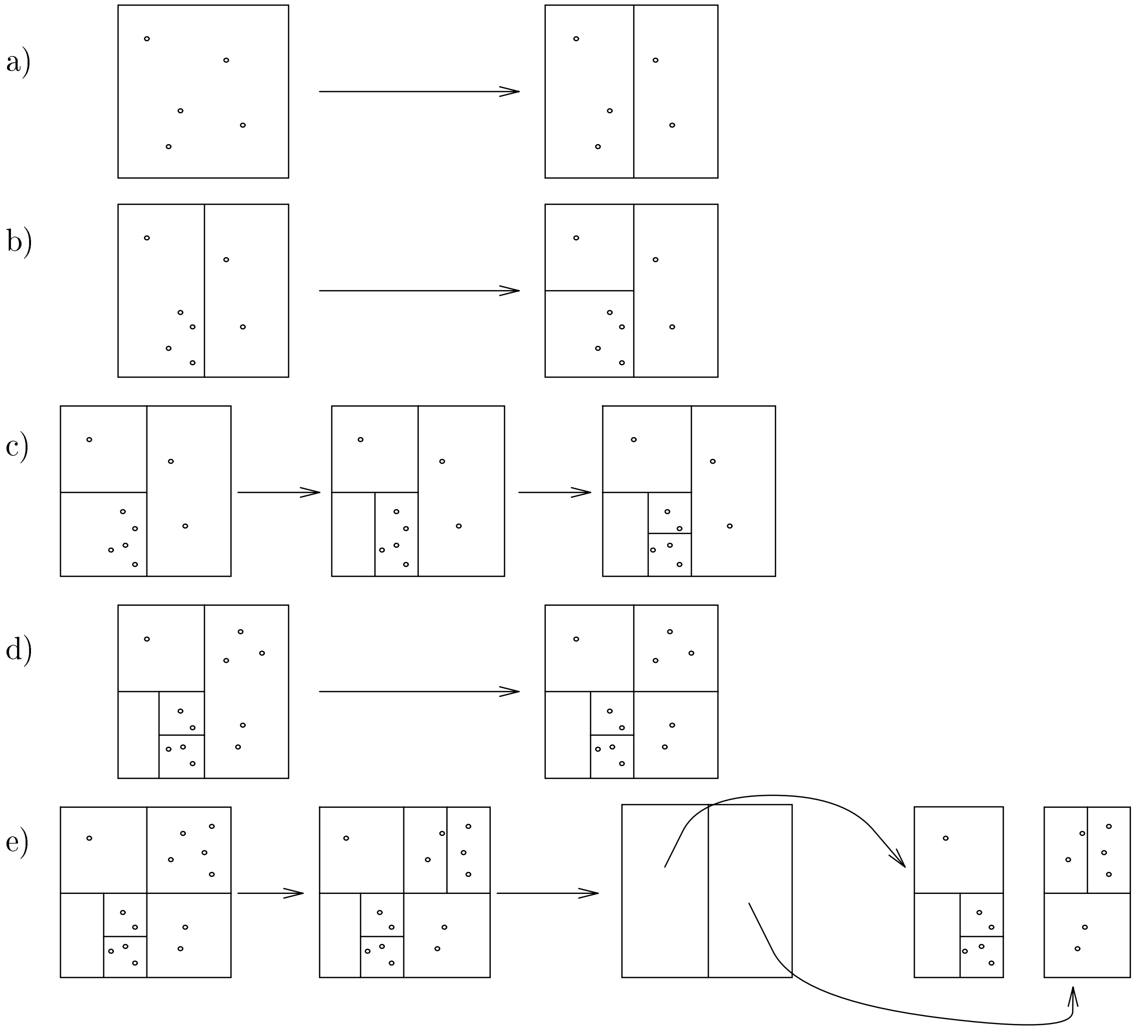


Abbildung 15: Dynamischer Aufbau des Buddy-Hashbaums

seinen linken unteren und rechten oberen Eckpunkt zu repräsentieren. Da eine große Auflösung des Datenraums D vorausgesetzt wird, wäre die Folge dieser Repräsentation eine geringe Kapazität der Directoryseiten. Dies würde zu einem schnellen Anwachsen des Hashbaums führen. Ähnlich zum Präfix B^+-Baum, BMEH, BANG-File und der Interpolations-Gitterdatei speichern wir nur soviele Präfix-Bits der Eckpunkte ab, wie zur Identifizierung der B-Rechtecke in einer Directoryseite notwendig sind. Somit ist jeder Datensatz wieder auffindbar.

Entsprechend zum BANG-File und der Interpolations-Gitterdatei benutzen wir hierbei die Interpolationsfunktion (IP-Funktion). Die IP-Funktion ist eine dynamische Hashfunktion, die unter anderem in [Bur 83] für mehrdimensionale lineare Hashverfahren vorgestellt wurde. Sie basiert auf dem Prinzip der Z-Ordnung [OM 84], bzw. dem Mischen von Bits.

Definition 2
Sei $w = d * z,\ z > 0$, *die Auflösung des Datenraums* D *und* $K = (K_1, \ldots, K_d) \in D$ *mit* $K_j = \sum_{i=1}^{z} b_i^j * 2^{-i}$, $1 \leq j \leq d$. *Für* $L \geq 0$ *ist der Z-Wert des Schlüssel* K *durch*

$$Z(K, L) := \lfloor 2^L * S(K) \rfloor$$

gegeben, wobei

$$S(K) := \sum_{i=1}^{z} \sum_{j=1}^{d} b_i^j * 2^{-d(i-1)-j}$$

der Mischwert des Schlüssels K *ist. Der Parameter* L *wird auch als Level bezeichnet.*

Die formale Definition 2 kann verbal einfacher beschrieben werden. Der Mischwert S(K) ist durch folgenden Bitstring gegeben: Das erste Bit ist das erste Bit der ersten Schlüsselkomponente, das zweite Bit ist das erste Bit der zweiten Schlüsselkomponente, usw. bis wir das d-te Bit gesetzt haben. Das (d+1)-te Bit ergibt sich aus dem zweiten Bit der ersten Schlüsselkomponente usw.. Die Mischfunktion S(K) wird in Abb. 16 in einfacher Weise veranschaulicht, wobei wir horizontal die Bitstrings und vertikal die Schlüsselkomponenten abgetragen haben. Für die Berechnung des Z-Werts werden nur die ersten L-Bits des Mischwerts berücksichtigt. Diese werden dann als Dezimalwert interpretiert. Man beachte, daß jede Achse gleichberechtigt ist und somit die Anzahl der Bits l_j, die die j-te Schlüsselkomponente für den Z-Wert Z(K,L) liefert, durch

$$l_j = \begin{cases} \lceil L/d \rceil & \text{falls } j \leq L \bmod d \\ \lfloor L/d \rfloor & \text{sonst} \end{cases} \tag{5}$$

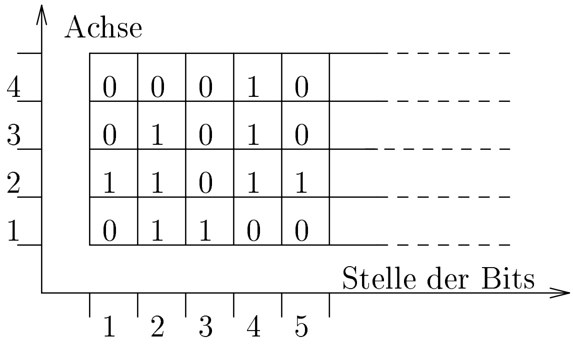
gegeben ist.

Definition 3
Sei $w = d * z,\ z > 0$ *die Auflösung des Datenraums* D *und* $K \in D$. *Die IP-Funktion* $h_L(K)$ *zum Level* L, $L \geq 0$, *ergibt sich aus*

$$h_L(K) := h(L, K) := \sum_{j=1}^{d} \sum_{i=1}^{l_j} b_i^j * 2^{d(i-1)+j-1} \tag{6}$$

wobei l_j *durch die Formel 5 gegeben ist. Wir bezeichnen* $h_L(K)$ *auch als IP-Wert des Schlüssels* K *(zum Level* L*).*

d =4, K = (0.01100..., 0.11011..., 0.01010..., 0.00010...)



$$\Rightarrow S(K) = 0.\mid 0100 \mid 1110 \mid 1000 \mid \ldots$$

Abbildung 16: Erzeugung des Mischwerts S(K) für einen 4-dimensionalen Schlüssel, der in binärer Darstellung gegeben ist

Die IP-Funktion zum Level L, $L \geq 0$, unterteilt den Datenraum $[0,1)^d$ mit Hilfe eines orthogonalen Gitters in gleichgroße Zellen. Während einer Verfeinerung des Gitters (Verdopplung der Anzahl von Zellen) wird jede Zelle mit Adresse i, $0 \leq i < 2^L$, in zwei Zellen unterteilt, wobei die linke Zelle die ursprüngliche Adresse und die rechte Zelle die Adresse $(i + 2^L)$ zugewiesen bekommt. Hierzu vergleiche man auch die Zuordnung von Adressen zu Zellen, wie sie in Abb. 17 für die Levels 3 und 4 aufgezeigt ist. Die Ähnlichkeit zwischen Z-Werten und IP-Werten eines Schlüssels $K \in D$ ist deutlich erkennbar. Ein Z-Wert, bzw. IP-Wert wird durch Umkehrung der Bits in den entsprechenden IP-Wert, bzw. Z-Wert transformiert.

Für die Darstellung der B-Rechtecke verwenden wir die IP-Werte der Eckpunkte der Rechtecke. Dabei ist das zur Berechnung benötigte Level durch die Feinheit des Gitters gegeben, so daß alle B-Rechtecke in einer Directoryseite eine eindeutige Darstellung besitzen. Betrachten wir die Situation aus Abb. 15 d, so können wir die B-Rechtecke durch

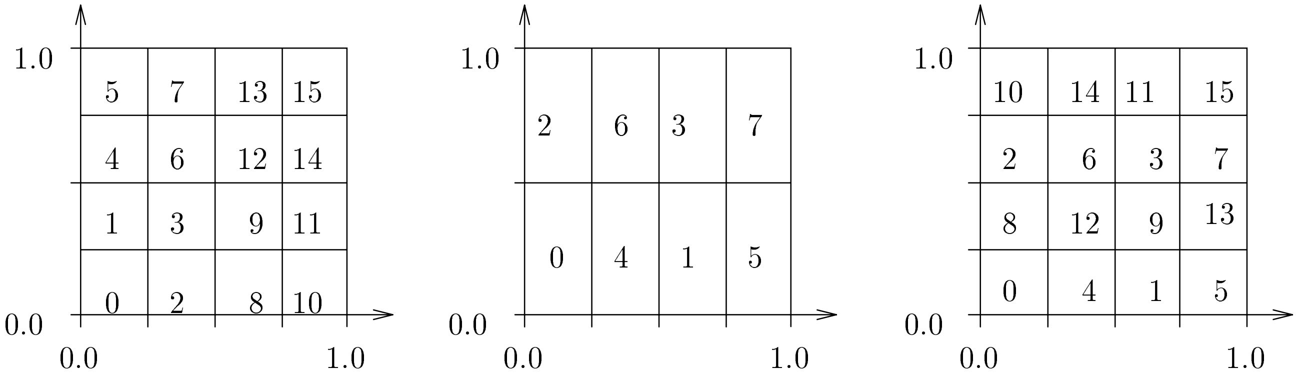


Abbildung 17: Die Adressen der Zellen des linken Gitter sind durch Z(K,4), die des mittleren Gitters durch $h_3(K)$ und die des rechten Gitters durch $h_4(K)$ berechnet worden

IP-Werte des Levels 4 eindeutig darstellen. In Abb. 18 haben wir die dazugehörige Directoryseite veranschaulicht. Zunächst wird in einer Directoryseite das zugehörige Level der Seite notiert. Die weiteren Einträge im Directory sind von der Form (ip_1,ip_2,adr), wobei ip_1 und ip_2 IP-Werte mit $ip_1 < ip_2$ sind und adr einer Seitenadresse entspricht. Die beiden IP-Werte ip_1 und ip_2 repräsentieren ein B-Rechteck, wobei ip_1 dem IP-Wert der linken unteren Zelle und ip_2 dem IP-Wert der rechten oberen Zelle entspricht. Im Gegensatz zum BANG-File ist die Reihenfolge der Einträge innerhalb einer Directoryseite nicht von Bedeutung.

Für die Adressierung der Zellen wird eine dynamische und nicht eine statische Hashfunktion benutzt. Das Problem für eine statische Hashfunktion ist, daß eine Verfeinerung oder Vergröberung des Gitters i. a. eine vollständige Neuadressierung der Zellen zur Folge hätte. Dies würde sich insbesondere negativ auf die benötigte CPU-Zeit unserer Verfahren auswirken. Die Operationen Verfeinern und Vergröbern eines Gitters werden aber sehr oft bei Hashbäumen benötigt. Im Gegensatz dazu wird z.B. beim Buddy-Hashbaum bei einer Verfeinerung des Gitters einer Directoryseite zu jedem IP-Wert, der einem rechten oberen Eckpunkt eines Rechtecks zugeordnet ist, nur ein konstanter Wert dazu addiert. Auch beim Splitten einer Directoryseite ist keine vollständige Neuberechnung notwendig, da für jeden IP-Wert nur eine Shift-Operation ausgeführt werden muß.

Das Level sowie die IP-Werte einer Directoryseite sind stets relativ zu dem Seitenbereich der Directoryseite und nicht global zum gesamten Datenraum D. Dadurch benötigt der Buddy-Hashbaum zur Repräsentation eines Eintrags i. a. nur wenige Bytes. Das Level einer Directoryseite kann kaum 31 überschreiten, so daß maximal 8 Bytes für die Beschreibung der Rechtecke und weitere 2 Bytes für die zugehörige Seitenadresse ausreichend sind. Da wir für jede Directoryseite das Level und die Anzahl der Einträge abspeichern, können wir, bei einer Seitengröße von 512 Bytes, mindestens 50 Einträge in einer Directoryseite unterbringen. Diese minimale Kapazität einer Directoryseite ist dabei unabhängig von der Dimension der Datensätze. Im Gegensatz dazu würden wir für d=2 und z=256, bei einer direkten Repräsentation der Rechtecke, 66 Bytes pro Eintrag benötigen, so daß nur sieben Directoryeinträge in einer Seite untergebracht werden könnten.

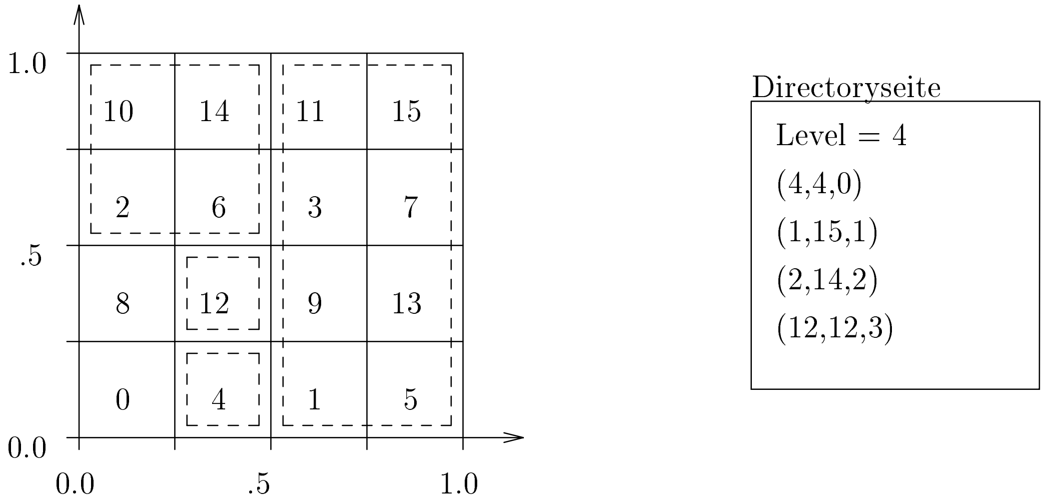


Abbildung 18: Partitionierung und Organisation einer Directoryseite des BHB

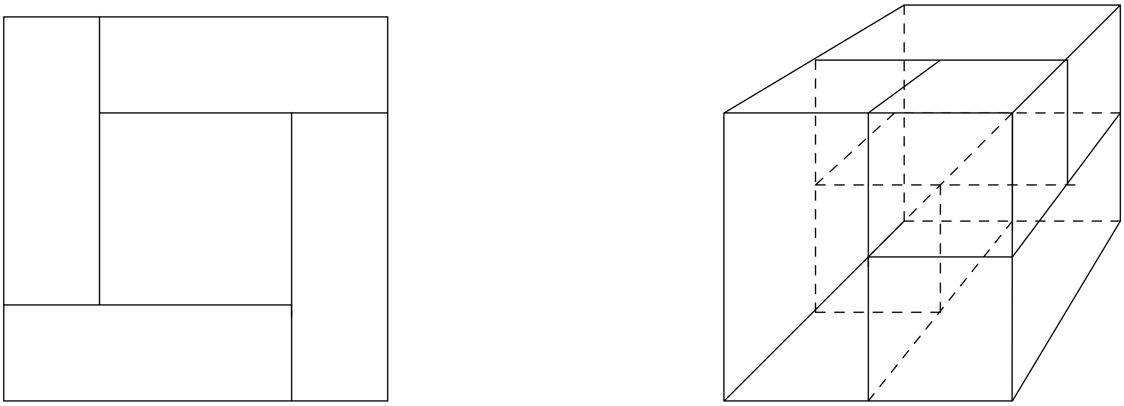


Abbildung 19: Verklemmungen für Seitenregionen, die auf der linken Seite für $d = 2$ Rechtecke und auf der rechten Seite für $d = 3$ B-Rechtecke sind

Im Gegensatz zur Gitterdatei oder dem erweiterbaren Hashbaum werden beim Buddy-Hashbaum Zeiger nicht mehrfach abgespeichert. Dadurch erwarten wir im Vergleich zu diesen Verfahren, insbesondere für sehr große Dateien (mit etwa 2^{20} Datensätzen), ein erheblich geringeres Anwachsen des Directories. Zudem werden durch die eindeutige Repräsentation von Zeigern kostspielige Reorganisationen vermieden. Da Seitenbereiche des Buddy-Hashbaums stets B-Rechtecke sind, können unterfüllte Seiten leicht miteinander verschmolzen werden.

Die von uns vorgeschlagene Repräsentation von Rechtecken erlaubt es, im Gegensatz zur Multi-Level Gitterdatei, nicht nur B-Rechtecke, sondern beliebig geformte Rechtecke, als Seitenbereiche zuzulassen. Dies kann insbesondere dann von Vorteil sein, falls eine Datei einen statischen Zustand erreicht hat und deshalb ein Packen von unterfüllten Seiten, deren gemeinsame Region einem Rechteck entspricht, die Span wesentlich erhöhen könnte. Würden wir für dynamische Dateien beliebig geformte Rechtecke als Seitenregionen zulassen, so könnte es leicht zu Verklemmungen in einer Directoryseite kommen. Betrachten wir hierzu die Partitionierung des Datenraums in fünf Regionen auf der linken Seite der Abb. 19, wobei die Seitenbereiche beliebige Rechtecke sein können. Per Definition ist ein Verschmelzen von zwei beliebigen Seiten nicht möglich. Die Vereinigung zweier Seitenbereiche ergibt stets einen nicht-rechteckigen Bereich. Der Begriff der "Verklemmung" ist hier nicht ganz zutreffend, da beim Buddy-Hashbaum durch Löschen aller Datensätze einer Seite die zugehörige Seitenregion nicht länger im Directory repräsentiert wird und somit solch eine "Verklemmung" aufgelöst werden kann. Wir sprechen deshalb hier von einer Pseudo-Verklemmung. Im Gegensatz zum BHB kann es bei der Gitterdatei, wegen der vollständigen Partitionierung des Datenraums, zu einer echten Verklemmungssituation kommen. Offensichtlich können Pseudo-Verklemmungen zu einer geringen Span des BHB führen.

Für $d > 2$ können auch bei Seitenbereichen, die B-Rechtecke sind, Pseudo-Verklemmungen auftreten, siehe rechte Seite der Abb. 19. Die in [Hin 85] vorgeschlagenen Verfahren zum Vermeiden einer Verklemmung bei der Gitterdatei lassen sich ebenfalls für den Buddy-Hashbaum anwenden. Dadurch können wir folgende, für das Splitten einer Directoryseite des Buddy-Hashbaums, wichtige Eigenschaft garantieren:

Eine Pseudo-Verklemmung von mehreren B-Rechtecken in einer Directoryseite

> kann nur dann verhindert werden, falls es mindestens für eine Achse eine Partitionierungslinie gibt, die senkrecht auf dieser Achse steht und die den Seitenbereich halbiert, ohne dabei ein B-Rechteck zu schneiden.

Da wir Pseudo-Verklemmungen stets vermeiden, können wir stets eine Partitionierungslinie finden, die den Überlauf einer Directoryseite behebt ohne dabei ein zugehöriges B-Rechteck dieser Seite zu schneiden. Damit werden vom Buddy-Hashbaum die Kriterien (K1) und (K2) gleichzeitig erfüllt.

3.3 Algorithmen

In diesem Abschnitt betrachten wir ausführlich die wichtigsten Algorithmen des Buddy-Hashbaums. Desweiteren stellen wir Varianten des Buddy-Hashbaums vor, die für eine verbesserte Leistung des Verfahrens sorgen. Die Algorithmen beschreiben wir ähnlich wie für PLOP-Hashing, je nach Komplexität in Modula-2 oder Umgangssprache. Dabei heben wir Namen von Prozeduren und neu definierte Datentypen stets durch Fettdruck aus dem Text hervor.

In den folgenden Algorithmen wird nicht explizit die Pufferorganisation des Buddy-Hashbaums erwähnt. Ähnlich zum B-Baum ist es für den Buddy-Hashbaum sinnvoll einen Puffer zu organisieren, in welchem alle Seiten eines Pfads des Hashbaums liegen. So setzen wir bei einem Split, bzw. Verschmelzen einer Datenseite stets voraus, daß gegebenenfalls die Vorgängerseiten im Puffer liegen.

Zunächst gehen wir hier auf die in den Algorithmen verwendeten Datentypen ein, wobei wir der Einfachheit halber ”dynamische Felder” als Typen zugelassen haben. Weiterhin setzen wir voraus, daß der Typ **KeyComponent** die Schlüsselkomponente und der Typ **PageAdr** eine Sekundärspeicheradresse spezifiziert. Die wichtigsten Typen sind wie folgt gegeben:

```
TYPE    DirEntry= RECORD
                     ll,ru: IPValue (* = CARDINAL *);
                     adr: PageAdr;
                  END;
        Key     = ARRAY [1..d] OF KeyComponent;
        Page    = RECORD
                     numb: CARDINAL;
                     CASE dir: BOOLEAN OF
                        TRUE : level: CARDINAL;
                               entries: ARRAY OF DirEntry; |
                        FALSE: keys: ARRAY OF Key
                     END;
                  END;
```

Der Typ **DirEntry** entspricht genau einem B-Rechteck in einer Directoryseite, wobei **ll** zur linken unteren Ecke und **ru** zur rechten oberen Ecke des B-Rechtecks gehört. Weiterhin ist **adr** eine zu dem B-Rechteck gehörende Sekundärspeicheradresse, die auf die zugehörige Seite verweist. Die Konstante d gibt die Dimension des Schlüssels an. Eine Seite besteht aus einem Zähler **numb**, der jeweils die Anzahl der Einträge wiedergibt sowie einem boolschen Wert **dir**, welcher anzeigt, ob eine Seite eine Directoryseite bzw. eine Datenseite ist. In Abhängigkeit von der Seitenart besteht der Rest der Seite aus einem Feld mit Schlüsseleinträgen oder aus Directoryeinträgen. Für eine Directoryseite wird zusätzlich das zugehörige Level abgespeichert.

Zunächst stellen wir den Algorithmus **BHB_EMQ** für eine exakte Suche vor. Dabei entspricht die Variable **root** der Wurzel des Buddy-Hashbaums. Weiterhin verwenden wir vier Prozeduren, deren Bedeutung wir hier nur kurz erläutern möchten. Die Prozedur **SearchEntry** sucht in einer Directoryseite nach einem B-Rechteck, welches eine Zelle mit IP-Wert hashadr enthält. Falls solch ein B-Rechteck existiert, wird der Index der zugehörigen Feldkomponente zurückgegeben. Andernfalls wird ein Wert kleiner null als Resultat zurückgegeben. Die Prozedur **SearchRecord** arbeitet entsprechend zu der Prozedur **SearchEntry**, nur mit dem Unterschied, daß eine Datenseite nach dem Schlüssel $(K_1, \ldots, K_d)$ durchsucht wird. Die Prozedur **ReadPage** holt zu einer vorgegebenen Adresse die zugehörige Seite vom Sekundärspeicher in den Hauptspeicher. Die Aufgabe der Prozedur **NormalizeKey** ist, den Schlüssel K, der in einem vorgegebenen B-Rechteck liegt, auf den d-dimensionalen Einheitswürfel $[0,1)^d$ zu transformieren.

```
Algorithmus BHB_EMQ( $(K_1, \ldots, K_d)$: Key) : BOOLEAN;
(* gegeben: $0 \leq K_i < 1.0,\ 1 \leq i \leq d$ *)
BEGIN
    page := root; $\tilde{K}$ := $(K_1, \ldots, K_d)$;
    LOOP
        IF page.dir THEN
            hashadr := h(page.level, $\tilde{K}$) (* := $h_{page.level}(\tilde{K})$ *)
            ind := SearchEntry(page.entries, hashadr);
            IF ind < 0 THEN (* kein B-Rechteck gefunden *)
                RETURN FALSE
            END;
            $\tilde{K}$ := NormalizeKey( $\tilde{K}$, page.entries[ind].ru, page.entries[ind].ll, page.level);
            ReadNewPage( page, page.entries[ind].adr)
        ELSE
            ind := SearchRecord( page.keys, $(K_1, \ldots, K_d)$);
            RETURN (ind $\geq$ 0)
        END;
    END;
END BHB_EMQ;
```

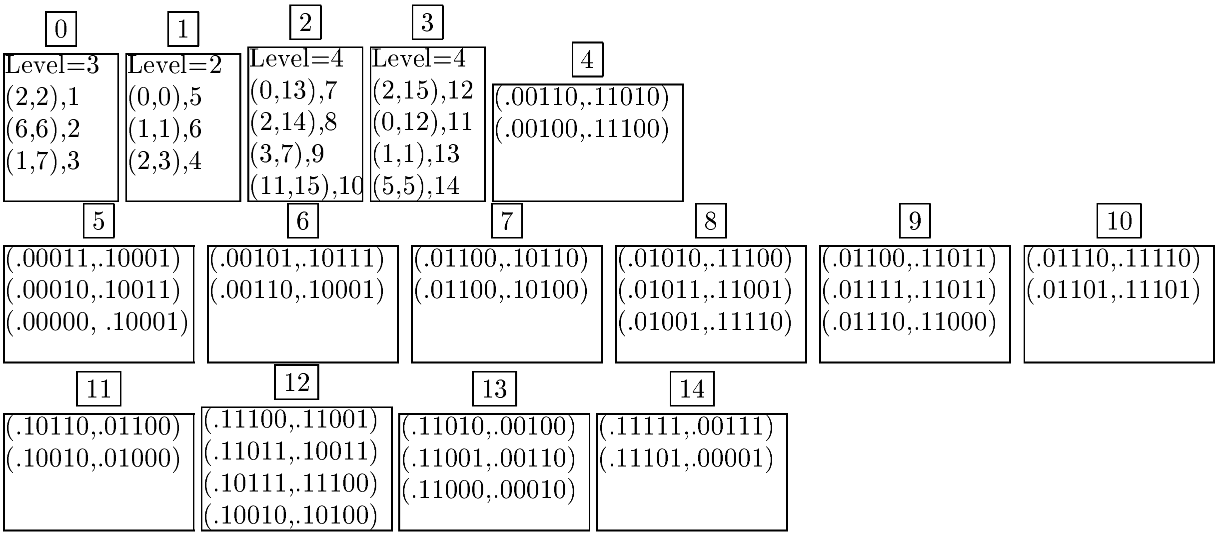


Abbildung 20: Struktur des Buddy-Hashbaums, Partitionierung des Datenraums und Aufbau der Datei nach dem Einfügen von 28 2-dimensionalen Datensätzen

Um den Algorithmus **BHB_EMQ** zu veranschaulichen, betrachten wir für $d = 2$ und $b = 4$ ein Beispiel, wobei wir $D = [0,1)^2$ mit Auflösung zehn voraussetzen. Nach Einfügen von 28 Datensätzen hat der BHB die Datei, wie in Abb. 20 zu sehen, organisiert. Bei der Suche des Datensatzes mit Schlüssel K=(.01011,.11001) gehen wir wie folgt vor:

- Zunächst bestimmen wir für den Schlüssel $\tilde{K}$ den IP-Wert bzgl. des Gitters der Wurzel. Somit erhalten wir hashadr $= h_3(\tilde{K}) = 6$. Die Zelle mit Adresse 6 liegt offensichtlich im B-Rechteck (6,6). Somit laden wir die Seite mit Adresse 2 in den Hauptspeicher und transformieren den Schlüssel $\tilde{K}$ bzgl. des B-Rechtecks (6,6). Wir erhalten somit $\tilde{K}$= (.011,.1001).

- Im zweiten Durchlauf der LOOP-Schleife berechnen wir wiederum den IP-Wert hashadr $= h_4(\tilde{K}) = 6$. Die Zelle mit Adresse 6 liegt nun im B-Rechteck (2,14). Sodann laden wir die Seite mit der dazugehörigen Adresse 8 in den Hauptspeicher.

- Schließlich finden wir in der Datenseite mit Adresse 8 den gesuchten Datensatz.

Die Algorithmen für eine exakte Suche und eine Bereichssuche unterscheiden sich nicht wesentlich. Bei der Bereichsanfrage müssen i. a. mehrere Pfade des Hashbaums durchlaufen werden. In jeder Directoryseite müssen alle B-Rechtecke gefunden werden, die die Anfrageregion schneiden. Dann müssen die zugehörigen Teilbäume des BHB nach Antworten durchsucht werden.

Der interessanteste, aber auch komplexeste Algorithmus des BHB ist der Einfüge-Algorithmus.

```
Algorithmus BHB_Insert( ($K_1, \ldots, K_d$): Key);
(* Vor.: $0 \leq K_i < 1.0$, $1 \leq i \leq d$ *)

BEGIN
  IF BHB_EMQ($K_1, \ldots, K_d$) THEN
    WriteErrorMsg(”Schlüssel existiert bereits in der Datei”)
  ELSE
    LOOP (* sei act nun die letzte im Puffer abgelegte Seite *)
      IF act.dir THEN (* exakte Suche endete in einer Directoryseite *)
        adr := $h_{act.level}(K)$
        IF (ein Buddy B zum B-Rechteck (adr,adr) existiert) AND
          (die zugehörige Seite zum Buddy B einen Datensatz aufnehmen kann) AND
          (die Situation nach einem Verschmelzen der Buddies B und (adr,adr)
          pseudo-verklemmungsfrei ist)
        THEN
          ersetze den Eintrag des Buddy B durch das minimal umgebende
          B-Rechteck der Buddies B und (adr,adr);
          setze act auf die Seite, die ehemals zum Buddy B gehörte;
        ELSE
          füge einen neuen Eintrag in die Seite act ein, wobei das B-Rechteck
          durch (adr,adr) gegeben ist und die Blockadresse auf eine neu erzeugte
          Datenseite verweist;
          IF (act ist überfüllt) THEN
            Split(act)
          END;
          setze act auf die neu erzeugte Datenseite;
        END;
      ELSE
        füge den Datensatz in die Datenseite act ein;
        IF (act ist überfüllt) THEN
          Split(act)
        END; EXIT
      END;
      MinimizeRegions();
    END (* LOOP *);
  END;
END BHB_Insert;
```

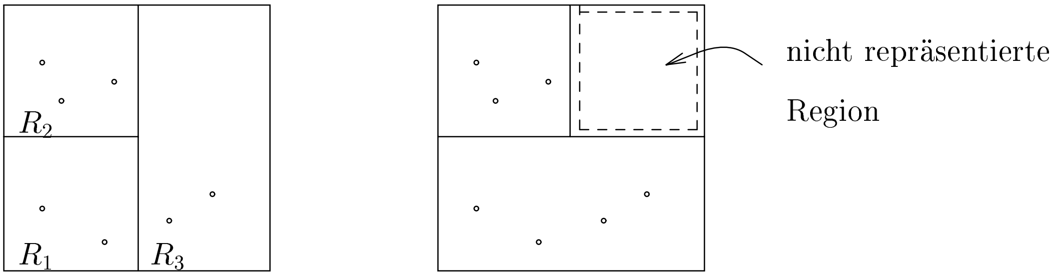


Abbildung 21: Partitionierung durch die Gitterdatei (links) und den Buddy-Hashbaum (rechts)

Insbesonders sind in diesem Algorithmus die Unterschiede des BHB zum Konzept der Multi-Level Gitterdatei zu erkennen. Der BHB ist *keine* balancierte Struktur, d. h. die Blätter des BHB liegen nicht notwendigerweise auf dem gleichen Level. Endet beim Einfügen die zugehörige exakte Suche in einem Directoryknoten und besteht nicht die Möglichkeit ein entsprechendes B-Rechteck zu finden, so wird *genau eine* Datenseite und nicht wie bei der Multi-Level Gitterdatei ein ganzer Pfad (bis zur Blattebene) an die Directoryseite gehängt. Dadurch garantiert der BHB, daß mindestens zwei Einträge in einer Directoryseite sind. Somit ist ein lineares Wachstum des Directory in der Anzahl der Datensätze gewährleistet.

Ein weiterer wesentlicher Unterschied zur Multi-Level Gitterdatei ist das Erzeugen von minimalen Seitenbereichen. Beim BHB verstehen wir unter der minimalen Region das kleinste Rechteck, das die zugehörigen Datensätze umgibt und zudem durch die Adreßfunktion berechenbar ist. Nach jedem Einfügen wird durch den Aufruf der Prozedur **MinimizeRegions** versucht, alle B-Rechtecke, die zu den Seiten des Puffers gehören, möglichst zu verkleinern. Der Vorteil kleinerer Seitenregionen besteht zum einen in der Reduzierung der Suchkosten, da z. B. erfolglose exakte Suchen frühzeitig abgebrochen werden. Zum anderen wird durch möglichst kleine Seitenbereiche die Span des BHB erhöht. Die Chance zu einem B-Rechteck einen Buddy zum Verschmelzen zu finden, ist bei kleinen B-Rechtecken höher als bei größeren B-Rechtecken. Betrachten wir dazu die Situation in Abb. 21. Im linken Teil ist die Partitionierung der Gitterdatei und im rechten die des BHB, jeweils für $b = 4$ und die gleiche Punktmenge ($n = 7$), aufgezeigt. Da das B-Rechteck R_3 der Gitterdatei beim BHB halbiert wird, ergibt sich die Möglichkeit die Seiten des reduzierten B-Rechtecks und des B-Rechtecks R_1 miteinander zu verschmelzen. Wir erhalten dadurch eine verbesserte Span.

Die Prozedur **MinimizeRegions** veranschaulichen wir hier kurz an Hand eines Beispiels. Wir nehmen dabei an, daß nach einer Insert-Operation drei Seiten im Puffer abgelegt worden sind, siehe Abb. 22. In der zweiten Directoryseite sind drei Regionen mit ihren zugehörigen Seitenadressen abgelegt, wobei die Seite das Level 3 hat. In der Datenseite liegen alle Datensätze in der linken unteren Ecke. Das zugehörige B-Rechteck könnte bzgl. jeder Achse einmal halbiert werden ohne daß Datensätze umgespeichert werden müssen. Der BHB halbiert das B-Rechteck aber nur bzgl. der x-Achse, da folgende Regel zu Grunde liegt:

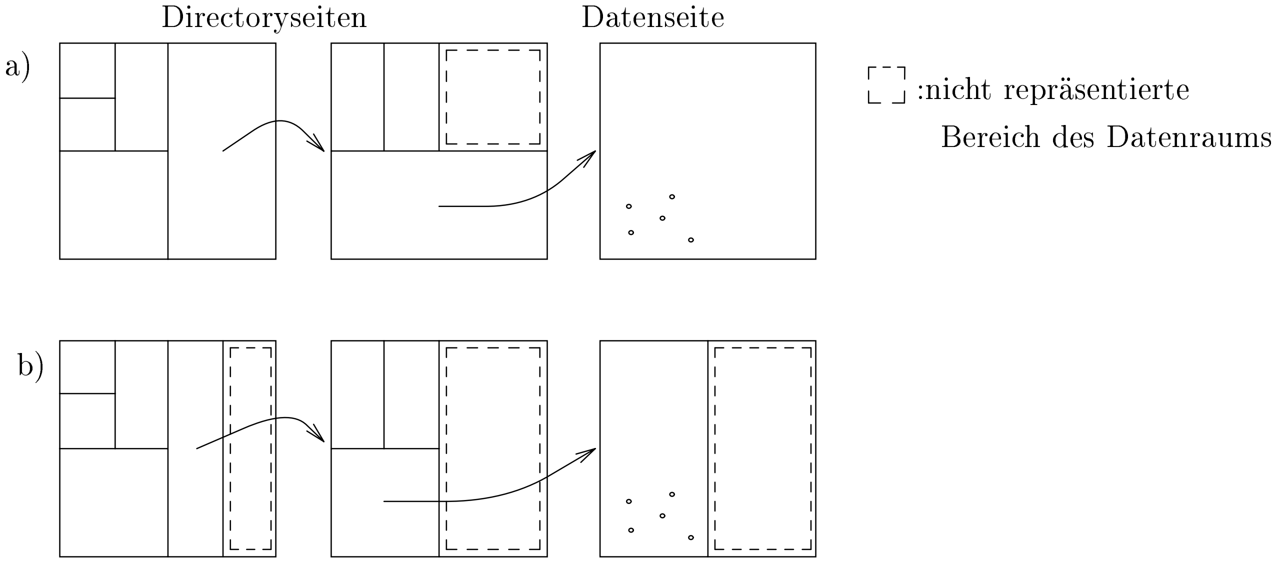


Abbildung 22: Reduzierung von Seitenregionen beim Einfügen eines Datensatzes durch den BHB, wobei in a) die Ausgangssituation und in b) die Endsituation aufgezeigt ist

> Reduziere ein B-Rechteck einer Seite nur solange, wie die darüberliegende Directoryseite das reduzierte B-Rechteck unter Beibehaltung des Levels adressieren kann.

Unter Berücksichtigung dieser Regel kann der Bereich der Datenseite nur bzgl. der x-Achse halbiert werden, da sonst das Level der darüberliegenden Directoryseite auf 4 erhöht werden müßte. Die Gründe für diese Beschränkung sind zum einen der erhöhte Aufwand beim Berechnen der IP-Werte bei einem hohen Level der Directoryseite. Zum anderen bringt das Erhöhen des Levels keine neue Möglichkeit, Seiten miteinander zu verschmelzen.

Der Buddy-Hashbaum unterstützt, ebenso wie PLOP-Hashing, einen dynamisch wachsenden und schrumpfenden Datenraum. Liegen alle Datensätze in einer Hälfte des Datenraums, so kann offensichtlich die Region der Wurzelseite (und somit der zu partitionierende Datenraum) halbiert werden. Andererseits kann der Datenraum durch Verdopplung beliebig oft erweitert werden. Hierfür ist nur eine Reorganisation der Wurzelseite nötig. Andere Teile des BHB sind von einer Erweiterung des Datenraums nicht betroffen. Im Gegensatz dazu ist bei der Gitterdatei oder beim BANG-File der zu partitionierende Datenraum fest vorgegeben und unveränderbar.

Eine weitere Möglichkeit zur Optimierung des BHB wurde in den bisherigen Algorithmen nicht berücksichtigt. Wir sind stets davon ausgegangen, daß Verweise auf Seiten, die im Directory gehalten werden, stets paarweise disjunkt sind. Dadurch sollte insbesondere das Wachstum des Directory reduziert werden. Diese Forderung erscheint aber genau dann zu restriktiv, falls es in einer Directoryseite mindestens zwei Verweise zu Datenseiten gibt, die weniger als 50 % gefüllt sind. Würden wir diese Datenseiten miteinander verschmelzen und die entsprechenden Zeiger redundant in der Directoryseite halten, so würde das Directory nicht vergrößert werden. Darüber hinaus würde diese Strategie, angewendet auf alle Datenseiten, eine Span von fast 50 % garantieren. Es könnte dabei nur genau einen Zeiger pro Directoryseite geben, der auf eine zu weniger als 50 % gefüllte Datenseite verweist. Der Mehraufwand für diese hohe Span muß beim Einfügen und Löschen von Datensätzen bezahlt werden. Es muß damit gerechnet werden, daß häufiger Seiten miteinander verschmolzen und gesplittet werden. Darüber hinaus kann bei einem Split einer Directoryseite nicht mehr garantiert werden, daß eine Partitionierungslinie gefunden wird, die den Seitenbereich halbiert und kein zugehöriges B-Rechteck schneidet. Ähnlich zum BANG-File kann das Problem der indirekten Splits auftreten, wobei die Folgeseiten gesplittet werden müssen, deren Seitenbereiche von der Partitionierungslinie geschnitten werden. Durch die Forderung von minimalen Seitenregionen rechnen wir i. a. nur mit wenigen indirekten Splits. Dennoch wird das Packen ausschließlich für Datenseiten erlaubt, damit diese von indirekten Splits betroffen sind und ein Einfügen auf einen Pfad des BHB beschränkt bleibt.

Neben der Span besitzt die gepackte Variante des BHB den Vorteil, Bereichsanfragen besser zu unterstützen. Da nur nah zusammenliegende Datenseiten gepackt werden, kann im Mittel mit weniger Zugriffen gerechnet werden.

Um so größer der Suchbereich der Anfrage ist, um so besser wirkt sich das Packen des BHB auf die Effizienz aus.

Das Packen der Datenseiten kann im Prinzip auf jede PZS angewendet werden. Da neben einer höheren Span die Leistung für Bereichsanfragen möglichst verbessert werden soll, ist nur das Packen von nah beieinanderliegenden Seiten sinnvoll. Dies kann gut von Hashbäumen unterstützt werden, während z. B. für die Gitterdatei es sehr aufwendig sein kann, zu einer unterfüllten Datenseite eine andere unterfüllte Datenseite zu finden, die in der Nähe liegt.

3.4 Analytische Betrachtungen

In diesem Abschnitt werden wir Leistungsmerkmale des Buddy-Hashbaums aufzeigen. Zunächst werden wir hier näher auf den worst-case für die Höhe des BHBs eingehen. Danach werden wir eine untere Schranke für die maximale Anzahl von Buddies herleiten. Dieser Parameter beeinflußt wesentlich das dynamische Verhalten des BHBs.

Die Leistung des BHB ist wesentlich durch seine Höhe h, $h \geq 0$, gekennzeichnet. Im schlechtesten Fall benötigt der BHB für Einfügen, Löschen oder (exaktes) Suchen O(h) Diskzugriffe.

Üblicherweise wird bei PZS und speziell bei Baumverfahren versucht, eine asymptotische Aussage mit Hilfe der O-Notation zu gewinnen. So ist z.B. wohl bekannt, daß die Höhe des B-Baums und des AVL-Baums in $O(\log n)$ anwächst. Im Gegensatz zum B-Baum ist der AVL-Baum aber nicht für die Organisation von Daten im Sekundärspeicher geeignet. Ein weiteres Beispiel dafür, daß die O-Notation teilweise nicht die gewünschte Aussagekraft besitzt, ist das für gleichverteilte Daten superlinear anwachsende Directory der 2-Level Gitterdatei [Reg 85]. Da die zugehörige Konstante sehr klein und gleichzeitig der Exponent nahe bei 1 liegt, ist für gleichverteilte Daten praktisch nichts von einem superlinearen Wachstum zu spüren. Würden wir nur mit Hilfe der O-Notation und den üblichen Annahmen die maximale Höhe des BHB bestimmen, so erhalten wir die Aussage, daß der Baum im schlechtesten Fall zu einer Liste degeneriert und deshalb die Höhe des BHB im schlechtesten Fall O(n) beträgt.

Im folgenden berücksichtigen wir für die Abschätzung der Höhe des Buddy-Hashbaums zwei praktisch relevante Nebenbedingungen:

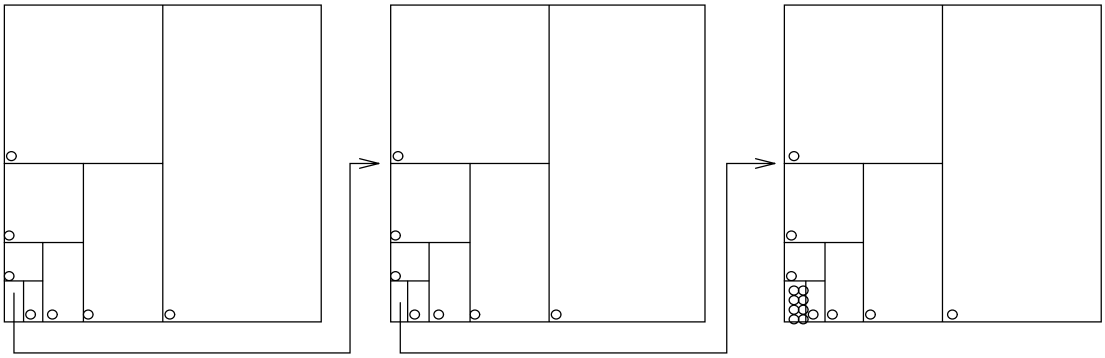


Abbildung 23: Degenerierung des BHB zu einer Liste (w = 27, b = 8)

- Die Auflösung des Datenraums D ist beschränkt. Im folgenden setzen wir $\mid D \mid = 2^w,\ w \geq d$, voraus.
- Die Anzahl der Datensätze in einer Datei ist beschränkt.

Zunächst wollen wir für den BHB aufzeigen, wann dieser zu einer Liste degenerieren kann. Wir gehen dabei von der Variante des BHB voraus, bei der die Datenseiten nicht gepackt werden. Dabei sei w, $w \geq d$, die Auflösung des Datenraums $D = [0,1)^d$ und $b = 2^c$ die Kapazität sowohl einer Daten- als auch einer Directoryseite.

Sei anfangs eine Datei mit genau einer leeren Datenseite gegeben. Zunächst werden b Datensätze $K^1, \ldots, K^b \in D$ in die Datei eingefügt, deren Mischwerte S(K^i), $1 \leq i \leq b$, in den ersten $w - c$ Bits identisch gleich 0 sind. Somit unterscheiden sich die Datensätze nur in den letzten c Bits und liegen deshalb in der linken unteren Ecke des Datenraums D eng beieinander. Sodann fügen wir w-c Datensätze $K^{b+1}, \ldots, K^{b+w-c}$ in die Datei ein, wobei bei dem Mischwert $S(K^{b+i})$ genau das i-te Bit 1, $1 \leq i \leq (w - c)$, und alle anderen Bits 0 gesetzt sind. Beim Einfügen eines Datensatzes muß stets ein neuer Eintrag in das Directory geschrieben werden, der auf eine neu bereitgestellte Datenseite verweist. Wir erhalten somit einen zu einer Liste degenerierten BHB, wie er in Abb. 23 veranschaulicht ist. Für $(b + w - c)$ Datensätze ist die maximale Höhe des BHB somit durch $\lceil (w - c)/b \rceil$ beschränkt.

Ein zweiter Spezialfall für den worst case tritt nach dem Einfügen von 2^w Datensätzen auf. In diesem Fall ist jeder mögliche Platz im Datenraum durch einen Datensatz belegt. Für die Höhe des BHBs ergibt sich dann $\lceil \frac{w-c}{c} \rceil$. Diese obere Schranke für die Höhe des BHBs ist bereits für den erweiterbaren Hashbaum [Oto 86] bekannt. Für diese PZS kann der worst case bereits nach dem Einfügen von (b+1) Datensätzen auftreten. Dagegen ist es für den BHB erforderlich, daß der Baum vollständig gefüllt ist. Nehmen wir an, daß es in diesem Fall 2 Einträge in der Wurzel des BHBs gibt, so dürfen in jedem Directoryknoten nur c Bits zur Adressierung der B-Rechtecke benutzt werden. Damit sind die B-Rechtecke in einer Directoryseite von gleicher Größe und entsprechen Zellen eines Gitters der Auflösung c. Damit nun keine nebeneinander liegende B-Rechtecke miteinander verschmolzen werden können, müssen beide zusammen mehr als zu 50% gefüllt sein.

Für den BHB ist insbesondere wichtig, daß es zu einem B-Rechteck viele Buddies gibt. Fällt ein Datensatz in nicht-partitionierten Datenraum, so wird ein möglichst kleines

B-Rechteck um diesen Punkt gelegt und anschließend ein Buddy zu diesem B-Rechteck gesucht.

Wir bezeichnen eine Menge von Rechtecken $\mathcal{R}$ als eine B-Partitionierung, falls

1. alle $S \in \mathcal{R}$ sind B-Rechtecke
2. $R \cap S = \emptyset$ für alle R, $S \in \mathcal{R}$

Im folgenden setzen wir eine B-Partitionierung $\mathcal{R}$ voraus. Weiterhin sei ein ganzzahliges $l > 0$ gegeben und es sei $L = l * d$, so daß durch die Hashfunktion h_L alle Regionen eindeutig berechenbar sind. Den kleinsten Wert von L bezeichnen wir auch als Level der B-Partitionierung. Das Volumen der kleinst möglichen Region ist hierbei durch 2^{-L} gegeben. Damit wir die maximale Anzahl von Buddies bestimmen können, wählen wir das Volumen der B-Rechtecke in $\mathcal{R}$ gerade minimal. Wir betrachten zunächst den Sonderfall für $d = 2$.

Theorem 1
Sei $\mathcal{R}$ eine 2-dimensionale B-Partitionierung mit Level $L\,(= 2 * l)$ und $S \in \mathcal{R}$ ein B-Rechteck mit Volumen 2^{-L}. Dann ist die maximale Anzahl von Buddies des B-Rechtecks S durch $(l + 1)$ gegeben.

Beweis:
Der Beweis gliedert sich in zwei Schritte. Zunächst zeigen wir, daß S nicht mehr als $(l+1)$ Buddies besitzen kann. In einem zweiten Schritt geben wir dann eine Situation an, in welcher die Region S genau $(l + 1)$ Buddies besitzt.

Mit Hilfe eines 2-dimensionalen Gitters der Auflösung L adressieren wir die B-Rechtecke der Partitionierung. Der Parameter l gibt dabei die Auflösung bzgl. einer Achse an. Nehmen wir nun ohne Beschränkung der Allgemeinheit an, daß S in der unteren linken Ecke des Datenraums liegt. Wir unterteilen den Datenraum $[0, 1)^2$ in $(l + 1)$ Gebiete $G_0, .., G_l$ mit $G_0 = [0, 2^{-l}) \times [0, 1)$ und $G_i = [2^{-l+i-1}, 2^{-l+i}) \times [0, 1)$ für $i \in \{1, .., l\}$. In jedem der Gebiete $G_0, .., G_l$ kann sich maximal ein Buddy des B-Rechtecks S befinden. Somit kann S nicht mehr als (l+1) Buddies besitzen.

Betrachten wir die B-Partitionierung $\mathcal{R} = \{S, R_0, ..., R_l\}$ mit

1. $S = [0, 2^l) \times [0, 2^l)$
2. $R_i = [2^{-i} - 2^{-l}, 2^{-i}) \times [2^{-l+i} - 2^{-l}, 2^{-l+i})$ mit $i \in \{0, .., l\}$

Die Behauptung ist, daß die B-Rechtecke R_i, $i \in \{0, .., l\}$ Buddies von S sind. Hierzu betrachten wir die B-Rechtecke

- $T_i = [0, 2^{-i}) \times [0, 2^{-l+i})$ mit $i \in \{0, .., l\}$

Trivialerweise gilt $T_i \supseteq R_i$ und $T_i \supseteq S$, $i \in \{0, .., l\}$.
Wir nehmen nun in Widerspruch zu unserer Behauptung an, daß $\mathcal{R}$ keine B-Partitionierung ist. Dann gibt es $i, j \in \{0, .., l\}$ mit $i \neq j$, so daß $T_i \cap R_j \neq \emptyset$. Auf Grund der Schnittbedingungen ergibt sich für die

1. Achse: $(0 < 2^{-j}) \wedge ((2^{-j} - 2^{-l}) < 2^{-i})$

2. Achse: $(0 < 2^{-l+j}) \wedge ((2^{-l+j} - 2^{-l}) < 2^{-l+i}$

Aus der 2. Gleichung ergibt sich $2^j - 1 < 2^i$ und somit folgt $j < i$. Andererseits ist dann die Gleichung $((2^{-j} - 2^{-l}) < 2^{-i})$ nur für $j \geq l$ und somit $i > l$ erfüllt. Die Bedingung $i > l$ steht dabei im Widerspruch zur unseren Annahme.

Theorem 2

Sei $\mathcal{R}$ eine B-Partitionierung mit Level L $(= d{*}l)$ und $S \in \mathcal{R}$ ein B-Rechteck mit Volumen 2^{-L}. Dann ist eine untere Schranke für die maximale Anzahl von Buddies des B-Rechtecks S wie folgt gegeben:

$$(l-1) * d * (d-1)/2 + d$$

Beweis:

Aus dem Spezialfall $d = 2$ ergibt sich, daß maximal 2 Buddies am Rand der 2-dimensionalen Ebene und die anderen $(l-1)$ Buddies im Inneren liegen. Da es in einem d-dimensionalen Raum mit $d \geq 2$ $d * (d-1)/2$ unabhängige Ebenen gibt und zudem pro Achse ein Buddy am Rand liegen kann, ergibt sich die zu beweisende Gleichung.

Eine weitere Forderung an PZS mit Directory ist ein lineares Anwachsen des Directory. Wie bereits im letzten Abschnitt erwähnt, werden beim BHB überflüssige Directoryseiten mit nur einem Eintrag vermieden. Somit ergibt sich für den BHB eine unbalancierte Baumstruktur, in der jede Directoryseite mindestens zwei Einträge enthält. Dadurch garantiert aber der BHB ein linear anwachsendes Directory.

Die Span ist ein wichtiges Kriterium für die Leistung einer PZS. Eine effiziente Beantwortung einer Bereichsanfrage mit einem großen Anfragebereich ist nur möglich, falls die zugehörige PZS eine hohe Span besitzt. Der BHB ohne Packen kann trotz des Konzepts der minimalen Regionen, keine hohe Span garantieren. So ergibt sich z.B. für den Fall, daß der BHB zu einer Liste degeneriert, eine Span, die kaum über 1/b liegt. Unser Vorschlag ist deshalb der gepackte BHB, d. h. unterfüllte Datenseiten, die zu einer Directoryseite gehören, können miteinander verschmolzen werden. Speziell für unsere "worst-case"-Verteilung würde sich für den gepackten BHB eine Span von bis zu (b-1)/b ergeben.

4 Vergleich von Punktzugriffsstrukturen

In den letzten zwei Abschnitten haben wir zwei neue mehrdimensionale PZS vorgestellt. Da eine genaue theoretische Analyse der Leistung des PLOP-Hashings und des Buddy-Hashbaums speziell für komplexere Anfragen nur unter sehr einschränkenden Voraussetzungen möglich ist, stellen wir in diesem Abschnitt einen experimentellen Vergleich dieser Verfahren vor. Aus Gründen der Vergleichbarkeit zu anderen PZS, haben wir die 2-Level Gitterdatei (2LGF) [Hin 85] in unsere Experimente mit einbezogen. Eine Implementierung der 2LGF wurde uns freundlicherweise von Dr. Hinrichs zur Verfügung gestellt.

Wir wollen hier nicht näher auf technische Details der Implementierungen eingehen und es deshalb bei folgenden Bemerkungen belassen:

- Alle Implementierungen sind in der Programmiersprache Modula-2 [Wir 85] erstellt worden. Die Implementierungen des Buddy-Hashbaums und des PLOP-Hashings sind mit einem kleinen "Host-Modul" versehen, in welchem die Rechnerabhängigkeiten spezifiziert sind.

- Die Implementierungen des Buddy-Hashbaums und des PLOP-Hashings sind sogenannte Prototyp-Implementierungen, während die Implementierung der 2LGF ein ausgereiftes Programmsystem darstellt. Da bisher bei diesen Implementierungen keine Optimierung der Hauptspeicheroperationen vorgenommen wurde, haben wir die benötigte CPU-Zeit der Verfahren nicht berücksichtigt. Wir möchten aber darauf hinweisen, daß bei großen Datenseiten die Suche eines Datensatzes in einer Datenseite nicht mehr vernachlässigbar ist gegenüber dem Lesen einer Seite vom Sekundärspeicher. Ein Ziel bei der Weiterentwicklung unserer Implementierungen ist deshalb eine effizientere interne Organisation der Directory- und Datenseiten. Im folgenden beurteilen wir die Effizienz eines Verfahrens durch die zugehörige Span und die Anzahl der Diskzugriffe für die jeweiligen Operationen (Einfügen, partielle Bereichsanfragen).

Unser Schwerpunkt bei den Experimenten liegt beim Buddy-Hashbaum. Zum einen haben wir einen ersten experimentellen Vergleich zwischen der 2LGF und dem PLOP-Hashing bereits in [KS 88] vorgestellt und zum anderen hat sich der BHB als das (über alle Experimente gemittelt) effizienteste und robusteste der drei Verfahren erwiesen. Wir möchten hier erwähnen, daß in [KSSS 89] ein experimenteller Vergleich des BANG-File [Fre 87], des hB-Baums [LS 87] und des Buddy-Hashbaums vorgestellt wurde. Erste Resultate dieses Vergleichs zeigen, daß selbst gegenüber diesen Verfahren der Buddy-Hashbaum die effizienteste und robusteste mehrdimensionale PZS ist.

Wie oben bereits erwähnt haben wir nur Prototypen der einzelnen Verfahren implementiert. Wegen der Komplexität der Algorithmen, haben wir einige Vereinfachungen bei unseren Verfahren vorgenommen.

Die Implementierung des PLOP-Hashings beinhaltet nicht die sehr wichtige interne Operation Anpassen, wie sie in Abschnitt 2.3.4 beschrieben wurde. Ohne diese Operation wirkt sich eine sortierte Eingabe der Daten negativ auf die Leistung des PLOP-Hashings aus. Eine Anpassung des orthogonalen Gitters an die Verteilung der Daten wird somit nur bzgl. der Splitachse (oder Mischachse) vorgenommen. Für unsere Experimente haben

Verteilung	$\emptyset$ Level	max. Level
Geo(0.1)	13	25
Geo(0.2)	11.5	16
Geo(0.3)	10.3	12
Kor(0.5	12.5	12
Kor(0.6)	7	10
Kor(0.7)	6.5	10
Kor(0.8)	6	8
Kor(0.9)	6	8
Kor(1.0)	6	8
reale Daten	7.5	10

Tabelle 2: Durchschnittliches Level ($\emptyset$ Level) und maximales Level der Directoryseiten des Buddy-Hashbaums bei verschiedenen Verteilungen

wir für PLOP-Hashing die Variante betrachtet, die die Kontrollfunktionen (Ex 1) und (Co 0.45) benutzt. Wir möchten hier bemerken, daß andere Kontrollfunktionen durchaus eine höhere Span und somit eine effizientere Abarbeitung von partiellen Bereichsanfragen bewirken können.

Für den Buddy-Hashbaum haben wir zwei wesentliche Vereinfachungen vorgenommen. Um die Organisation der Directoryseiten zu vereinfachen, haben wir das maximale Level einer Directoryseite durch 16, bzw. 31 beschränkt. Ein IP-Wert wurde dabei stets durch 2, bzw. 4 Bytes repräsentiert. Insgesamt benötigen wir zur Darstellung eines Eintrags im Directory (B-Rechteck und zugehöriger Seitenadresse) 6, bzw. 10 Bytes. Durch die Festlegung der für einen Eintrag zu reservierenden Bytes wird aber die Kapazität einer Directoryseite künstlich reduziert. In der Tab. 2 haben wir deshalb das maximale Level einer Directoryseite und das durchschnittliche Level aller Directoryseiten für alle Experimente notiert, nachdem wir alle Datensätze in die Dateien eingefügt hatten. Auf die verschiedenen Verteilungen, die wir bei den Experimenten betrachtet haben, werden wir später noch ausführlich eingehen. In einer weitergehenden Implementierung des Buddy-Hashbaums planen wir, zur Darstellung eines IP-Werts (zum Level L) genau $\lfloor L/8 \rfloor$ Bytes zu verwenden. Wir erwarten dadurch eine höhere Kapazität der Directoryseiten und einen höheren Verzweigungsgrad im Hashbaum, wodurch insbesondere die Höhe des Directory reduziert wird.

Im folgenden lassen wir für den Buddy-Hashbaum als Seitenregionen nur B-Rechtecke zu, obwohl unsere Darstellung der Seitenregionen auch beliebig geformte Rechtecke erlaubt. B-Rechtecke könnten entsprechend wie bei der Multi-Level Gitterdatei [WK 85] dargestellt werden, was i. a. zu einer nochmaligen Erhöhung der Kapazität einer Directoryseite führt.

Bisher ist für den Buddy-Hashbaum das Packen von Datenseiten noch nicht implementiert. Trotzdem kann durch einfache Simulation der Effekt des Packens auf die Leistung des BHB sichtbar gemacht werden. Auch kann in einfacher Weise die Leistung bei Anfragen für den gepackten BHB aufgezeigt werden. Das Packen des BHB wird rein sequentiell durchgeführt. Sind mehr als zwei Datenseiten unterfüllt, so ist die Entscheidung, welche der Seiten zusammen gepackt werden, unabhängig von den zugehörigen Seitenregionen. Würde man nun versuchen, die Seiten zu packen, deren Seitenregion möglichst nah beieinander liegen, so wäre mit einem verbesserten Anfrageverhalten zu rechnen. Trotzdem fällt bei unseren Experimenten der z. T. große Leistungsunterschied zwischen dem gewöhnlichen und dem gepackten BHB auf. In weiteren Untersuchungen wollen wir überprüfen, ob ein effizienteres Packen der Seiten die Leistung einer Bereichsanfrage verbessern kann, ohne dabei wesentlich die Einfügekosten zu erhöhen.

Weiterhin haben wir in der jetzigen Implementierung des BHB die Minimierung von Regionen in den Einfüge-Algorithmus noch nicht eingebaut. Stattdessen werden Regionen nur beim Einfügen eines Datensatzes minimiert, wenn die entsprechende Seite gesplittet wird.

Bei bisherigen mehrdimensionalen PZS verschlechtert sich i.a. die Leistung bei nicht gleichverteilten Daten im Vergleich zu gleichverteilten Daten erheblich. Unser primäres Ziel ist deshalb, das Verhalten der PZS bei nicht-gleichverteilten Daten aufzuzeigen. Andere Parameter wie die Kapazität einer Datenseite oder die Dimension eines Schlüssels haben dagegen bei allen PZS eine ähnliche Wirkung. So verbessern sich z. B. bei jeder Zugriffsstruktur die Antwortzeiten bei einer Bereichsanfrage, falls die Kapazität einer Datenseite erhöht wird. Wir haben deshalb in allen Experimenten die Kapazität einer Datenseite (b=50), die Größe einer Seite (512 Bytes), sowie die Dimension der Schlüssel (d=2) konstant gehalten. Zudem haben wir in den Experimenten als Datenraum stets den 2-dimensionalen Einheitswürfel $[0,1)^2$ vorausgesetzt.

Für Schlüsselkomponenten $K \in [0,1)$ betrachten wir im wesentlichen drei Verteilungen. Wir setzen dabei $K = \sum_{i\geq 1} b_i * 2^{-i}$ voraus. Weiterhin sei zu einem Ereignis X, Pb(X) die Wahrscheinlichkeit mit der das Ereignis X eintrifft. Die in unseren Experimenten vorkommenden Verteilungen sind nun wie folgt gegeben:

1. Eine Schlüsselkomponente $K \in [0,1)$ folgt einer Gleichverteilung, falls

$$Pb(a \leq K < b) = \frac{1}{b-a} \quad \forall a, b \in [0,1), a < b$$

2. Eine Schlüsselkomponente $K \in [0,1)$ folgt einer geometrischen Verteilung mit Parameter λ, falls

$$Pb(b_i = 0) = \lambda \quad Pb(b_i = 1) = 1 - \lambda \quad \forall i \geq 1$$

   Wir bezeichnen diese Verteilung kurz mit Geo(λ).

3. Sei $S = (S_1, S_2) \in [0,1)^2$ ein zweidimensionaler Schlüssel, wobei jede Komponente *unabhängig* gleichverteilt ist. Ein Schlüssel (K_1, K_2) folgt einer korrelierten Gleichverteilung Kor(c) mit Parameter c, $0.5 \leq c \leq 1.0$, falls (K_1, K_2) durch

$$\begin{aligned} K_1 &= c * S_1 + (1-c) * S_2 \\ K_2 &= (1-c) * S_1 + c * S_2 \end{aligned}$$

   gegeben ist. Seien zwei korrelierte Verteilungen Kor(c_1), Kor(c_2) mit $0.5 \leq c_1 < c_2 \leq 1.0$ gegeben, so sagen wir, daß die Verteilung Kor(c_1) stärker korreliert ist als die Verteilung Kor(c_2).

Abbildung 24: Auf der linken Seite der Abb. ist die Verteilung der realen Daten zu erkennen, während auf der rechten Seite die zugehörige Partitionierung des BHB aufgezeigt ist.

Für jedes unserer Verfahren haben wir insgesamt neun Dateien mit je 100,000 Datensätzen und eine Datei mit 81,750 Datensätzen erzeugt. Die Datensätze unserer ersten sechs Dateien folgen einer korrelierten Gleichverteilung Kor(c) mit $c = 1.0, 0.9, \ldots, 0.5$. Diese Testreihe zeigt insbesondere auf, wie sich die Abhängigkeit der Schlüsselkomponenten auf die Leistung unserer Verfahren auswirkt. In einer zweiten Testreihe haben wir drei Dateien erzeugt, in denen die einzelnen Schlüsselkomponenten unabhängig voneinander einer geometrischen Verteilung Geo(λ) mit $\lambda = 0.3, 0.2, 0.1$ folgen. Schließlich haben wir in unserer zehnten Datei reale Daten eingefügt, die wir freundlicherweise von dem Landesvermessungsamt Nordrhein-Westfalen zur Verfügung gestellt bekommen haben. In der Abbildung 24 haben wir die Daten und die zugehörige Partitionierung des BHB im Fall der realen Daten aufgezeigt. Diese 2-dimensionalen Punktdaten wurden aus Höhenlinien gewonnen und sortiert in die Datei eingefügt.

Die Leistung der Verfahren beim Aufbau der Dateien, in denen die geometrisch verteilten Datensätze abgespeichert sind, haben wir für die 2LGF und den Buddy-Hashbaum kurz in Tab. 3 aufgezeigt. Hieraus ist ersichtlich, daß beide Verfahren im Durchschnitt nur wenige Zugriffe beim Einfügen eines Datensatzes benötigen. Wegen der Beschränkung auf ein 2-Level Directory und unter der Annahme, daß die Wurzel vollständig im Hauptspeicher gehalten werden kann, benötigt die 2LGF im Durchschnitt weniger Zugriffe als der Buddy-Hashbaum. Dagegen ist die maximale Zugriffszeit beim Buddy-Hashbaum niediger als bei der 2LGF. Für den gepackten Buddy-Hashbaum haben wir zu den Einfügekosten keine Messungen vorgenommen. Wir erwarten aber keine wesentliche Erhöhung der Einfügekosten im Vergleich zum gewöhnlichen Buddy-Hashbaum.

Mehr von Interesse als die Aufbauzeit einer Datei sind die Zugriffszeiten für (partielle)

Verteilung	Verfahren	Diskzugriffe
Geo(0.3)	2LGF	3.072
	BHB	3.007
Geo(0.2)	2LGF	3.072
	BHB	3.775
Geo(0.1)	2LGF	3.732
	BHB	3.789

Tabelle 3: Durchschnittliche Anzahl der Diskzugriffe (gemittelt über die letzten 50,000 Einfügungen) für eine Einfüge-Operation bei der 2LGF und beim BHB

Bereichsanfragen. Wir haben deshalb für alle Dateien durch jedes unserer Verfahren 100 Anfragen beantworten lassen und dabei die Anzahl der benötigten Zugriffe gemessen. Unter einer quadratischen Bereichsanfrage verstehen wir hierbei eine Bereichsanfrage mit quadratischem Suchbereich. Wir haben dabei folgende fünf Typen von Anfragen betrachtet:

(A1) gleichverteilte quadratische Bereichsanfrage mit Flächeninhalt 0.001

(A2) gleichverteilte quadratische Bereichsanfrage mit Flächeninhalt 0.01

(A3) gleichverteilte quadratische Bereichsanfrage mit Flächeninhalt 0.1

(A4) partielle Anfrage, bei welcher die erste Achse spezifiziert ist

(A5) partielle Anfrage, bei welcher die zweite Achse spezifiziert ist

Für unsere Dateien haben wir pro Anfragetyp (A1) - (A5) jeweils 20 Anfragen betrachtet. Die Anzahl der Diskzugriffe für die Beantwortung der Anfragen haben wir in den Tabellen 4, 5 und 6 aufgezeigt. Wir haben in den Tabellen jeweils die absolute Anzahl von Diskzugriffen notiert, da unabhängig von der jeweiligen PZS für die oben erwähnten Verteilungen dieselben Daten erzeugt und dieselben Anfragen gestellt wurden. Wir haben somit bei unseren PZS jeweils dieselben Antwortmengen erhalten. Die Anzahl der Antworten ist für jede Datenverteilung in unseren Tabellen aufgezeigt. Da wir meist zufällig erzeugte Daten und zufällig erzeugte Anfragen betrachten, ist die Anzahl der Antworten (Trefferquote) bei einer partiellen Anfrage sehr gering. Aus diesem Grund haben wir die Trefferquote bei partiellen Anfragen nicht in unserer Statistik berücksichtigt. Zusätzlich zur Anfrageleistung haben wir in den Tabellen 4, 5 und 6 für jedes Verfahren die durchschnittlichen Kosten bei einer exakten Suche (emq), die Span nach dem Aufbau der Datei und für Verfahren mit Directory das Verhältnis von Directory- zu Datenseiten (dpd) notiert. Noch zu bemerken bleibt, daß wir für PLOP-Hashing keine Anfragen für die Verteilung Kor(0.5) gestartet haben, da die zu erwartende Leistung schon aus der exakten Zugriffszeit und der Span schon klar ersichtlich ist.

	Korrelierte Gleichverteilungen								
Verteilung		emq	Span	dpd	(A1)	(A2)	(A3)	(A4)	(A5)
Kor(0.5)	Antworten				0	32,687	334,594		
	2LGF	2	35%	7%	52	2,271	19,878	370	415
	PLOP	14	41.3%						
	BHB	2	70.4%	1.7%	3	906	9,642	40	79
	gepackter BHB	2	72.8%	1,8%	3	879	9,362	40	79
Kor(0.6)	Antworten				2,305	20,717	295,645		
	2LGF	2	61.5%	1.6%	177	902	10,388	693	914
	PLOP	2.1	46.8%		256	1,311	13,288	1,416	1,921
	BHB	3	63.0%	2.0%	145	831	10,169	579	737
	gepackter BHB	3	66.2%	2.1%	139	805	9,781	574	731
Kor(0.7)	Antworten				2,260	24,444	299,606		
	2LGF	2	61.5%	1.2%	184	1,018	10,498	782	1,369
	PLOP	1.5	49.1%		240	1,286	12,100	1,311	1,762
	BHB	2	63.0%	2.0%	175	1,005	10,386	640	1,315
	gepackter BHB	2	66,3%	2.1%	174	979	9,997	633	1,300
Kor(0.8)	Antworten				2,212	22,733	276,029		
	2LGF	2	73.3%	1.5%	165	883	8,230	837	1,157
	PLOP	1.3	55.2%		224	1,047	10,498	1,257	1,792
	BHB	2	75.0%	1.7%	154	844	8,137	929	1,064
	gepackter BHB	2	77.0%	1.7%	154	837	8,009	923	1,061
Kor(0.9)	Antworten				2,023	21,328	232,876		
	2LGF	2	62.4%	0.95%	183	1,069	8,431	1,069	1,312
	PLOP	1.3	68.0%		234	1,186	9,308	1,166	2,002
	BHB	2	66.3%	1.8%	169	998	7,901	1,072	1,186
	gepackter BHB	2	71.2%	1.9%	169	963	7,603	1,064	1,181
Kor(1.0)	Antworten				1,954	19,939	201,321		
	2LGF	2	70.1%	1.1%	155	870	6,666	970	1,440
	PLOP	1.01	76.3%		152	772	5,894	1,257	1,310
	BHB	3	70.1%	2.2%	166	892	6,715	1,048	1,440
	gepackter BHB	3	74.2%	2.2%	166	874	6,472	1,042	1,429

Tabelle 4: Leistungsvergleich bei korrelierten Gleichverteilungen

Die Leistungsdaten für korrelierte Gleichverteilungen sind in der Tabelle 4 aufgeführt. Bei Betrachtung der Ergebnisse können wir folgendes bemerken:

- PLOP-Hashing reagiert am empfindlichsten auf korrelierte Daten. Bei unabhängiger Gleichverteilung der Daten, die der Kor(1.0)-Verteilung entspricht, ist PLOP-Hashing den anderen Verfahren überlegen. Je stärker die Daten korreliert sind, um so schlechter ist die Leistung des PLOP-Hashings.
- Der Buddy-Hashbaum und die 2LGF besitzen für die Daten, die einer Kor(1.0), Kor(0.9), Kor(0.8) und Kor(0.7)-Verteilung folgen, annähernd dieselbe Leistung. Für die Kor(0.6)-verteilten Daten benötigt der Buddy-Hashbaum etwa 5-20 % weniger Zugriffe bei den Anfragen. Für diese Verteilung macht sich zum erstenmal bemerkbar, daß der Buddy-Hashbaum leere Bereiche des Datenraums, im Gegensatz zur Gitterdatei, im Directory nicht repräsentiert.
- Um so mehr der Parameter der korrelierten Verteilung sich 0.5 nähert, um so größer wird der Leistungsunterschied des Buddy-Hashbaums zur 2LGF. Für die Kor(0.5)-verteilten Daten, benötigt der BHB für die Bereichsanfragen nur noch in etwa die Hälfte der Zugriffe, die die 2LGF für dieselben Anfragen benötigt. Für partielle Anfragen benötigt der BHB bis zu neunmal weniger Diskzugriffe. Mit einer wachsenden Anzahl von Datensätzen wird der Leistungsunterschied bei diesen Verfahren bedingt durch das schnelle Anwachsen des Directory der Gitterdatei größer.
- In Tabelle 4 ist kein Unterschied in der Leistung des gepackten Buddy-Hashbaum zum nicht-gepackten Buddy-Hashbaum festzustellen.

Insgesamt gesehen geht aus der Testreihe mit den korrelierten Daten der BHB als Sieger hervor. Beachten wir die Resultate für extrem korrelierte Daten nicht, so besitzt die 2LGF ebenfalls eine akzeptabele Leistung. Dagegen bietet PLOP-Hashing nur für gleichverteilte und schwach korrelierte Daten eine zu den anderen Verfahren vergleichbare Leistung.

Bei exakten Anfragen benötigt der BHB je nach Verteilung drei bzw. zwei Zugriffe. Dies ergibt sich daraus, daß bei etwa 100000 Datensätzen, unabhängig von der Verteilung, die Höhe des BHB von zwei auf drei ansteigt. Die großen Schwankungen bei der Span des BHB oder der 2LGF sind dadurch erklärbar, daß, bedingt durch die Halbierungsstrategie beim Splitten von Datenseiten, bei einer Gleichverteilung der Daten viele Datenseiten in etwa gleichzeitig überlaufen und geplittet werden müssen. In Abb. 25 haben wir die Entwicklung der Span des BHB bei der Kor(1.0)-Verteilung aufgezeigt. Hierbei ist klar das zyklische Verhalten der Span zu erkennen. Im Durchschnitt erhalten wir für diese Verteilung eine Span von 69%.

Betrachten wir nun Tabelle 5, in welcher die Leistung für die geometrisch verteilten Daten aufgezeigt ist. Die Resultate können wir folgendermaßen zusammenfassen:

- Die Span der 2LGF und des gewöhnlichen BHB liegen deutlich unter dem Wert, der bei den korrelierten Daten erreicht wurde. Dagegen besitzt der gepackte BHB eine unverändert hohe Span und beantwortet so Bereichsanfragen wesentlich effizienter als die anderen Verfahren.
- Die geometrisch verteilten Daten entsprechen, für einen genügend kleinen Parameter, etwa dem schlechtesten Fall des BHB, wie er in Abschnitt 3.4 vorgestellt wurde.

Abbildung 25: Entwicklung der Span in Abhängigkeit der Anzahl der Datensätze

	Geometrische Verteilungen								
Verteilung		emq	Span	dpd	(A1)	(A2)	(A3)	(A4)	(A5)
Geo(0.3)	Antworten				3,143	7,313	153,313		
	2LGF	2	61.7%	1.6%	219	422	5,771	905	1,337
	PLOP	1.23	58,2%		304	570	7,279	1,088	1,723
	BHB	3	61.3%	2.0%	218	422	5,790	686	1,196
	gepackter BHB	3	74.4%	2.4%	212	407	5,081	680	1,179
Geo(0.2)	Antworten				1,267	10,372	66,183		
	2LGF	2	50.4%	2.1%	150	626	3,232	1,160	1,225
	PLOP	1.33	54.7%		183	657	3,489	1,142	1,694
	BHB	3	51.1%	4.4%	177	663	3,359	888	1,108
	gepackter BHB	3	74.4%	5.8%	166	556	2,658	846	1,062
Geo(0.1)	Antworten				677	2,364	24,742		
	2LGF	2	37.3%	4.0%	123	365	1,794	1,437	1,359
	PLOP	1.35	53.3%		149	312	1,593	1,013	1,573
	BHB	3	42.2%	5.3%	127	295	1,649	630	850
	gepackter BHB	3	69.7%	8.4%	111	238	1,212	579	786

Tabelle 5: Leistungsvergleich für geometrisch verteilte Daten

	Reale Daten							
	emq	Span	dpd	(A1)	(A2)	(A3)	(A4)	(A5)
Antworten				1,702	19,264	176,300		
2LGF	2	67.1%	1.5%	151	865	6,102	895	1,267
PLOP	1.8	47.3%		203	1,093	7,191	2,102	2,347
BHB	2	67.1%	1.7%	150	881	6,116	940	1237
gepackter BHB	2	72.3%	1.8%	149	854	5,838	929	1219

Tabelle 6: Leistungsvergleich für reale Daten

- Das Verhältnis von Directory- zu Datenseiten ist beim BHB stark angestiegen. Zu berücksichtigen ist hierbei, daß nur bei Daten, die einer Geo(0.2) und einer Geo(0.1)-Verteilung folgen, vier Bytes für einen IP-Wert belegt wurden. In allen anderen Experimenten haben wir jeweils für einen IP-Wert zwei Bytes reserviert.

Trotz des prozentualen hohen Aufwands für das Directory besitzt der gepackte BHB die beste Leistung für Daten, die einer geometrischen Verteilung folgen. Bemerkenswert ist, daß der Leistungsunterschied zwischen der 2LGF und dem gepackten BHB bei partiellen Anfragen wesentlich größer ausfällt als bei quadratischen Bereichsanfragen.

In unserem letzten Experiment haben wir reale, aus Höhenlinien erzeugte, Daten, durch die verschiedenen PZS organisieren lassen. Offensichtlich sind diese Daten noch so verteilt, daß der BHB, der gepackte BHB und die 2LGF in etwa die gleiche Leistung aufweisen. Die relativ schlechte Leistung des PLOP-Hashing ergibt sich durch die sortierte Eingabe der Daten.

In allen Experimenten hat der gepackte BHB die beste Leistung aufgezeigt. Die Span des BHB lag stets über 66 %, in den meisten Experimenten sogar über 70 %. Dagegen degeneriert die Leistung der 2LGF für extrem schiefe Verteilungen wie der Kor(0.5)-Verteilung oder der Geo(0.1)-Verteilung. Für eine größere Anzahl von Datensätzen erwarten wir, daß die 2LGF die Bedingung, das Wurzeldirectory ganz im Hauptspeicher zu halten, nicht mehr erfüllen kann. Speziell für Daten, die nur einen kleinen Teil des Datenraums belegen, scheint der BHB geradezu ideal zu sein. Durch die Eigenschaft, leere Bereiche nicht im Directory zu repräsentieren und zusätzlich Seitenbereiche möglichst klein zu halten, können Anfragen sehr gut unterstützt werden. Auch die Forderung vieler PZS nach einem rechteckigen Datenraum ist für den Buddy-Hashbaum nicht von Bedeutung. Im Prinzip wirkt sich die Form des Datenraums nicht auf die Leistung aus.

Der einzige Kritikpunkt für den BHB könnte das schlechte Verhältnis von Datenseiten zu Directoryseiten bei den geometrischen Verteilungen sein. Wir möchten hier aber nochmals erwähnen, daß die Geo(λ)-Verteilung für kleine Werte von λ dem schlechtesten Fall des BHB nahe kommt.

5 Raumzugriffsstrukturen

Wie bereits in der Literatur mehrfach aufgezeigt wurde, sind einfache Punktzugriffsstrukturen (PZS) i. a. für Anwendungen wie CAD, automatisiertes Erstellen von Karten oder Bildverarbeitung nicht geeignet. Insbesondere benötigen wir zur Abspeicherung von Raumdaten (Polygone, Rechtecke,...) geeignete Zugriffsstrukturen, die wir im folgenden als Raumzugriffsstrukturen (RZS) bezeichnen. Hauptziel aller RZS ist es, nah beieinander liegende Objekte auch physisch nah beieinander abzuspeichern. Somit sollen Anfragen, die die Geometrie und Lage eines Objekts betreffen, möglichst effizient beantwortet werden. Die Beschreibung eines komplexen Raumobjekts kann beliebig lang sein und sich insbesondere über mehrere physische Seiten erstrecken. Dagegen haben wir bei PZS stets vorausgesetzt, daß ein Punktobjekt genau einer Seite zugeordnet werden kann.

5.1 Klassifikation und Anforderungen

Im folgenden verstehen wir unter einem d-dimensionalen Raumobjekt eine endliche Vereinigung von zusammenhängenden Mengen, die aus unendlich vielen d-dimensionalen Punkten, $d \geq 1$, zusammengesetzt sind. Weiterhin nehmen wir an, daß alle d-dimensionalen Raumobjekte im d-dimensionalen Einheitswürfel $[0,1)^d$ liegen. Wie bereits bei PZS erwähnt, kann dies durch eine einfache Transformation der Objekte erreicht werden.

Die bekannteste Methode Raumobjekte in heutigen Datenbanken zu organisieren, ist die Randdarstellung (boundary representation) [Req 80]. Hierbei sind Raumobjekte durch Beschreibung ihrer Randobjekte gegeben. Betrachten wir hierzu das Beispiel zur Abspeicherung von 2-dimensionalen Polygonen, so wird jedes Polygon durch seine Linienzüge, die Linienzüge durch die Eckpunkte, die Eckpunkte durch x- und y-Koordinate beschrieben. Dabei wird z. B. bei relationalen Datenbanken, ein Objekt auf mehrere Dateien verteilt, wobei diese Dateien im wesentlichen die Beziehung der i-dimensionalen Raumobjekte untereinander beschreiben, $0 \leq i \leq d$. Zusätzlich erfordert die Abbildung etwas komplexerer Raumobjekte eine Erweiterung eines relationalen DBS, wie sie z. B. in [Mei 86] vorgeschlagen wurde.

Will man in einem relationalen DBS zu einem vorgegebenen Objektidentifikator auf die Geometrie und Lage eines Objekts zugreifen, so werden i. a. hierfür mehrere Verbund-Operationen benötigt. Deshalb sind Anfragen zur Geometrie und Lage der Objekte offensichtlich nur mit hohem Aufwand zu beantworten. Um solche Operationen effizienter zu unterstützen, wird eine RZS benötigt, die unabhängig von der exakten Repräsentation (die z. B. in Randdarstellung vorliegt) zusätzlich geometrische Informationen der Objekte schnell zugreifbar organisiert.

Die prinzipielle Idee einer RZS ist das Raumobjekt durch eine einfache Überdeckung zu approximieren. Diese Approximationen der Raumobjekte sind vergleichbar mit den Adreßtabellen und Directories der üblichen PZS. Sie dienen primär dazu, die Suche bzgl. ausgewählter Merkmale der Raumobjekte effizienter durchzuführen. Folgende allgemeinen Anforderungen sollte dabei eine RZS erfüllen:

Anforderung 1 die Überdeckung eines Raumobjekts soll möglichst exakt dem Raumobjekt entsprechen

Anforderung 2 die Menge der zusätzlich abgespeicherten Informationen sollte wesentlich kleiner sein als die Menge der Daten, die für eine exakte Beschreibung des Raumobjekts notwendig sind

Anforderung 3 eine RZS soll sowohl die Suche bzgl. geometrischen als auch nicht-geometrischen Attributen effizient unterstützen

Anforderung 4 die abgespeicherte Überdeckung eines Raumobjekts soll eindeutig sein

Bei einer Raum-Anfrage, wie z. B. *Suche alle Raumobjekte, die ein vorgegebenes Rechteck schneiden*, wird in zwei Schritten vorgegangen:

1. suche alle Raumobjekte (Kandidaten) deren approximative Überdeckungen die Anfrage erfüllen
2. für jedes Kandidatenobjekt muß mit Hilfe der exakten Darstellung überprüft werden, ob es die Anfrage tatsächlich erfüllt

Je genauer die Überdeckung dem Raumobjekt entspricht, umso kleiner wird die zu untersuchende Kandidatenmenge und der Aufwand in Schritt 2 sein (siehe Anforderung 1). Andererseits kann Schritt 1 nur schnell ausgeführt werden, wenn die Beschreibung der Überdeckung nicht zu komplex ist (siehe Anforderung 2).

Die Anforderung 3 ergibt sich aus der Beobachtung, daß die Suche von Raumobjekten sich nicht notwendigerweise nur auf geometrische Attribute bezieht. So könnte z. B. bei der Organisation von Polygonen, neben Anfragen nach dem Ort des Polygons, die Anzahl der Eckpunkte eines Polygons ein wesentliches Suchkriterium bei Anfragen sein. Deshalb sind RZS, die die Anforderung 3 erfüllen, auch für die Organisation von nicht-geometrischen komplexen Objekte geeignet.

Das Testen auf Gleichheit von zwei Raumobjekten, bzw. von zwei Überdeckungen ist speziell in CAD-Anwendungen eine wichtige Operation. Darüberhinaus muß beim Einfügen eines Raumobjekts dieser Test vorgenommen werden. Entsprechend den Suchoperation empfiehlt es sich, zunächst den Identitätstest auf den Überdeckungen der Raumobjekte ablaufen zu lassen. Durch eine eindeutige Darstellung der Überdeckung (Anforderung 4) ist garantiert, daß nur bei Gleichheit der Überdeckungen die zugehörigen Objekte gleich sein können.

In Abhängigkeit der Komplexität der Überdeckungen unterscheiden wir zwei Klassen von RZS:

Klasse 1 Verfahren, die zwar ein Raumobjekt durch ein einfacheres Objekt überdecken, aber die Beschreibung der Überdeckung sich wiederum über mehrere physische Seiten erstrecken kann

Klasse 2 Verfahren, die ein d-dimensionales Raumobjekt durch das minimal umgebendes Rechteck (MUR) überdecken, dessen Seiten parallel zu den Achsen des Datenraums liegen

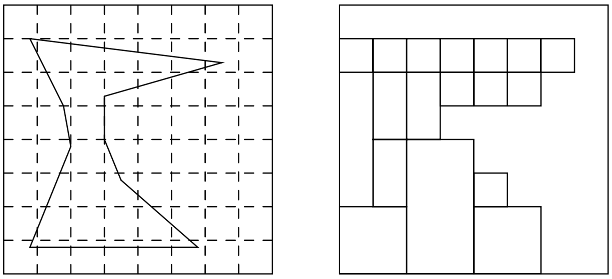


Abbildung 26: Rasterapproximation durch ein 8x8-Gitter

Innerhalb des PROBE-Projekts wurde eine RZS der Klasse 1 speziell für die Organisation von Raumobjekten im Sekundärspeicher entwickelt [OM 88]. Dabei wird ein Objekt durch ein orthogonales Gitter überdeckt, das den Objektraum in Zellen gleicher Größe unterteilt. Alle die von diesem Objekt geschnitten Zellen werden in einer Datei abgelegt. Wie in der Bildverarbeitung sprechen wir hierbei von einer (konservativen) Rasterapproximation. Eine äußerst kompakte Darstellung der Rasterapproximation kann erreicht werden, wenn Zellen zu größeren Rechtecken mit Hilfe von Z-Werten [OM 84] zusammengefügt werden. In Abb. 26 haben wir eine Situation veranschaulicht, in welcher wir ein nicht konvexes Objekt mit Hilfe eines 8×8-Gitters approximiert haben. Das Objekt wird zwar von 30 Zellen des Gitters geschnitten, es genügen aber 14 Z-Werte um die Rasterapproximation zu beschreiben. Die eindimensionalen Z-Werte können hierbei sehr effizient durch einen B^+-Baum organisiert werden. Ein großer Vorteil der Rasterapproximation ist, daß vor allem die in geographischen und kartographischen Anwendungen benötigte Operation der Überdeckung von verschiedenen Karten effizient unterstützt werden kann. Trotzdem kann die Anzahl der abgespeicherten Z-Werte in Abhängigkeit der Auflösung des zu Grunde liegenden Gitters sehr hoch sein. Betrachten wir dazu unser Beispiel aus Abb. 26, so müssen bei einem Raumobjekt, das durch seine 9 Eckpunkte beschrieben werden kann, bereits 14 Z-Werte abgelegt werden. Dieses Verhalten widerspricht unserer Anforderung 2. Insbesondere kann, wenn mehrere neue Objekte eingefügt werden, die dieses Objekt schneiden, der Effekt auftreten, daß diese 14 Z-Werte nicht notwendigerweise in einer Seite abgelegt werden. Insbesondere wirkt sich dies negativ auf Schritt 1 von Raum-Anfragen aus.

Eine weitere RZS dieser Klasse ist der Zellbaum [Gün 88]. Der Zellbaum überdeckt d-dimensionale Raumobjekte durch eine Vereinigung von konvexen Raumobjekten (Zellen). Diese d-dim. konvexen Zellen werden in einem k-d-B-Baum organisiert, in welchem auch nicht orthogonale Partitionierungslinien erlaubt sind, siehe Abb. 27. Ähnlich zum R^+-Baum müssen konvexe Zellen zerschnitten werden, wenn sie von einer Partitionierungslinie geschnitten werden. Somit erfüllt auch der Zellbaum die Anforderung 2 nicht, da dieser Effekt nicht von der Komplexität der Objekte, sondern von dem "Überlappungsgrad der Objekte" im Datenraum abhängt. In Zusammenhang mit dem R^+-Baum werden wir in Abschnitt 5.4.1 näher auf dieses Problem eingehen. Die Anforderung 1 wird von dem Zellbaum besonders gut erfüllt, da i. a. das exakte Raumobjekt im Zellbaum abgelegt wird. Soll der Zellbaum die Anforderung 4 erfüllen, so muß bei einer Änderung eines bereits eingefügten Raumobjekts, dieses vollständig aus dem Zellbaum entfernt werden, dann eine neue konvexe Zerlegung des Objekts berechnet werden und schließlich die konvexen Teilobjekte neu eingefügt werden. Zu beachten ist, daß beim Zellbaum, entsprechend dem

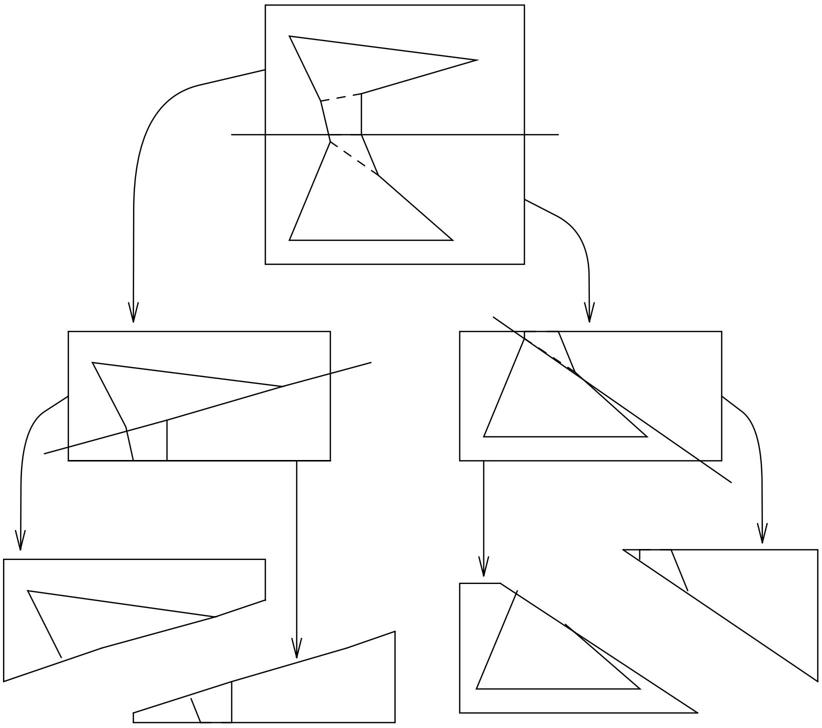


Abbildung 27: Struktur eines binären Zellbaums

k-d-B-Baum, das Löschen von Objekten und die damit verbundenen Reorganisationen der Baumstruktur einen hohen Aufwand erfordern.

Typische RZS der Klasse 2 sind der R-Baum [Gut 84], R^+-Baum [SRF 87], Box-Excell [MT 83], sowie diverse Erweiterungen von mehrdimensionalen PZS ([NH 85], [SW 88], [SK 88]). Obwohl diese Überdeckung mit MUR die Raumobjekte nur sehr grob annähert und somit Information verloren geht, bleiben die wichtigsten geometrischen Eigenschaften, nämlich die Ausdehnungs- und Orts-Attribute, erhalten. Unsere Anforderung 1 wird aber von diesen RZS i. a. nicht erfüllt. Dagegen können RZS, wie z. B. der R-Baum, die Anforderung 2 erfüllen, da nur eine konstante Anzahl von Bytes pro Raumobjekt zusätzlich abgespeichert wird. In Abschnitten 5.3 und 5.4 werden wir uns ausführlich mit diesen RZS auseinandersetzen. Interessant zu bemerken ist, daß in dem Projekt "Automatisierte Liegenschaftskarte" verschiedener Landesvermessungsämter ein ähnlicher Ansatz gewählt wurde [Stö 87]. Damit räumliche Information von Objekten ohne Ausführung komplexer Operationen zugreifbar wird, wurde eine sogenannte Objektkoordinate für jedes Objekt berechnet.

5.2 Architektur eines Anfrageprozessors

Aufgabe eines Anfrageprozessors für Raumobjekte ist es, die einzelnen Dateien, auf welche die Raumobjekte verteilt sind, und die dazugehörigen Zugriffsstrukturen effizient zu organisieren und zu koordinieren. Wir können einen Anfrageprozessor als eine "komplexe" Zugriffsstruktur für Raumobjekte ansehen. Die Notwendigkeit für einen Anfrageprozessor ergibt sich aus unseren Beobachtungen, daß RZS der Klassen 1 und 2 nur teilweise unsere Anforderungen 1 und 2 erfüllen. Für sehr komplexe Objekte ist der Verlust an Informationen bei MUR sehr hoch. Insbesondere stellt sich oft erst beim Überprüfen der exakten Darstellung des Objekts heraus, daß das Objekt eine Anfrage nicht erfüllt. Gerade für sehr komplexe Objekte ist dies mit hohem Aufwand verbunden. Es wäre also in diesen Fällen vorteilhaft, Raumobjekte zusätzlich durch eine RZS der Klasse 1 zu verwalten, so daß möglichst oft vermieden wird auf die exakte Darstellung eines Objekts zuzugreifen. Dagegen genügt für relativ einfache Objekte eine Approximation durch MUR. Die endgültige Überprüfung mit Hilfe der exakten Darstellung, ob ein einfaches Raumobjekt die Anfrage erfüllt, erfordert nur einen geringen Aufwand. Dieser Aufwand kann geringer sein als der, der für die Überprüfung mit Hilfe einer RZS der Klasse 1 notwendig ist.

In Abb. 28 haben wir die Struktur des Anfrageprozessor dargestellt. Der Anfrageprozessor besteht aus drei Schichten. Dabei werden in den ersten zwei Schichten nur Überdeckungen der Raumobjekte gehalten, während auf der dritten Schicht das Raumobjekt exakt repräsentiert wird. Auf die dritte Schicht des Anfrageprozessors gehen wir hier nicht näher ein, da diese sehr stark von der jeweiligen Architektur des jeweiligen DBS abhängt. Wie oben bereits erwähnt, nehmen wir an, daß unsere Objekte exakt in Randdarstellung gegeben sind. In [KW 87] ist aufgezeigt, wie in verschiedenen DBS Raumobjekte in Randdarstellung abgebildet werden können.

Auf der zweiten Schicht des Anfrageprozessors werden Rasterapproximationen der Raumobjekte organisiert. Im Gegensatz zum PROBE-Vorschlag [OM 88] ordnen wir dabei die Rasterapproximation eines Objekts i. a. einer Datei oder einem zusammenhängenden Speicherbereich zu, so daß ein schneller Zugriff auf die gesamte Approximation eines Objekts gewährleistet ist. Auf der ersten Schicht organisieren wir die MUR der Raumobjekte. Wir können die ersten zwei Schichten des Anfrageprozessors als das Directory des Prozessors ansehen, das primär die Aufgabe hat, zu einer vorgegebenen Anfrage eine möglichst kleine Obermenge der Antworten zu berechnen.

Um das Vorgehen einer Anfrage zu verdeutlichen, betrachten wir das Beispiel einer Rechteck-Schnitt Anfrage. Für solch eine Anfrage suchen wir alle Raumobjekte, die ein vorgegebenes Rechteck schneiden. Zunächst suchen wir auf der ersten Schicht des Prozessors nach allen Raumobjekten, deren MUR das Anfragefenster schneiden. Für all diese Raumobjekte, die eine genügend komplexe Struktur besitzen, testen wir auf der zweiten Schicht, ob die zugehörigen Rasterapproximationen das Anfragefenster schneiden. Dadurch wird sich i. a. die in der ersten Schicht bestimmte Kandidatenmenge nochmals reduzieren. Auf der dritten Schicht testen wir schließlich für alle Raumobjekte aus der Kandidatenmenge mittels der exakten Darstellung, ob das Anfragefenster geschnitten wird.

Wie in Abb. 28 veranschaulicht, erwarten wir einen geringeren Speicherplatzbedarf für die Überdeckungen im Vergleich zu der exakten Darstellung. Um dies zusätzlich zu garantieren, speichern wir nur die Rasterapproximationen der Objekte ab, die eine gewisse Komplexität aufweisen. Dagegen wird stets das MUR der Raumobjekte in der ersten Schicht abgelegt. Zu jedem MUR gibt es einen Zeiger, der direkt auf die nächst genauere Darstellung verweist. Um möglichst viele Diskzugriffe zu vermeiden, sollte diese Darstellung auf möglichst wenige physische Seiten verteilt sein. Ein effizienter räumlicher Zugriff ist in unserem Anfrageprozessor somit nur über die MUR der Objekte möglich.

Ein Grund, daß wir uns für eine Rasterapproximation der Raumobjekte entschieden haben, ist die graphische Ausgabe der Antworten einer Anfrage. Im Prinzip genügt es

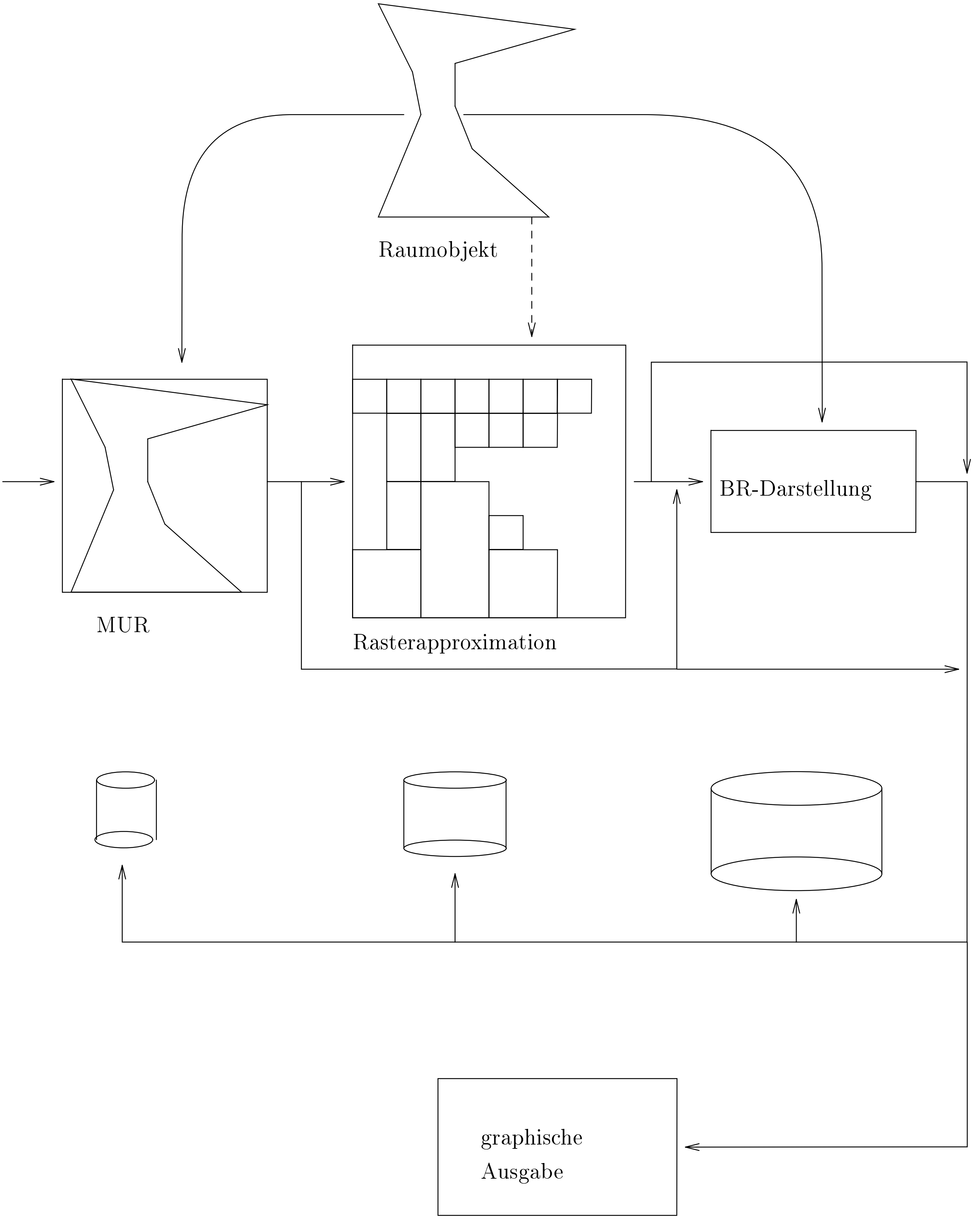


Abbildung 28: Drei-Schichten Organisation eines Anfrageprozessors

für die graphische Ausgabe die Rasterapproximation der Raumobjekte zu benutzen. Ein Benutzer wird i. a. nicht an der reinen ASCII-Darstellung seiner Objekte interessiert sein. Diese Darstellung ist dann sinnvoll, wenn ein durch eine Anfrage erzeugtes Objekt sofort wieder in die einzelnen Dateien abgelegt wird. Darüber hinaus sind in der praktischen Anwendung, Benutzer teilweise nicht an einer exakten Beantwortung der Anfrage interessiert, sondern erwarten nur einen kurzen Überblick möglicher Antworten. Deshalb kann in solch einem Fall das explizite Testen mit Hilfe der exakten Repräsentation, ob das Objekt die Anfrage erfüllt, entfallen.

Ein weiterer Grund für eine Rasterapproximation ist die effiziente Unterstützung der Operation *räumlicher Verbund*. Darunter verstehen wir die Berechnung des Schnittobjekts zu zwei vorgegebenen Raumobjekten, die in einem gemeinsamen Datenraum liegen. Diese Operation wird insbesondere in kartographischen Anwendungen benötigt. Dabei sind zwei Karten vorgegeben, die verschiedene Eigenschaften einer Landschaft repräsentieren. Als Ergebnis wird eine neue Karte erzeugt, die die Gebiete kennzeichnet, die Eigenschaften gemeinsam besitzen. Beispielsweise könnten in einer Karte die Waldgebiete und in einer anderen Karte die Höhenlinien einer Region abgelegt sein. In einer neuen Karte sollen alle Waldgebiete abgespeichert werden, die über 500 Meter Höhe liegen.

5.3 Anforderungen an Strukturen basierend auf MUR

Unsere bisherigen Anforderungen an RZS haben wir sehr allgemein gehalten. In diesem Abschnitt werden wir RZS der Klasse 2 genauer betrachten und dabei unsere Anforderungen genauer spezifizieren.

Sei $\{O_1, \ldots, O_n\}$ eine Menge von d-dimensionalen Raumobjekten mit $O_i \subseteq D$, $1 \leq i \leq n$, wobei $n \geq 1$ die zeitabhängige Anzahl der Objekte ist. Wir setzen voraus, daß jedes Objekt O_i durch einen Objektidentifikator OId_i eindeutig bestimmt ist, $1 \leq i \leq n$. Wir betrachten im folgenden die Datei $\mathcal{R} = \{(R_1, OId_1), \ldots, (R_n, OId_n)\}$, wobei R_i das MUR von dem Objekt O_i ist ($R_i = Mur(OId_i)$), $1 \leq i \leq n$. Da nicht notwendigerweise ein MUR eindeutig das zugehörige Objekt bestimmt, wird zur Abspeicherung der Rechtecke ein Objektidentifikator benötigt. Ziel ist es eine Zugriffsstruktur zur effizienten Organisation der Datei $\mathcal{R}$ zu finden. Wenn wir von Effizienz sprechen, so beziehen wir uns, wie bei PZS auf die Anzahl der Sekundärspeicherzugriffe bei bestimmten Operationen. Hierbei sollen insbesondere folgende Suchoperationen (Raum-Anfragen) effizient unterstützt werden:

1. Punkt-Anfrage:
   Gegeben sei ein Punkt $P \in D$. Finde alle Datensätze (R,OId) in der Datei $\mathcal{R}$ mit $P \in R$.

2. Rechteck-Schnitt-Anfrage:
   Gegeben sei ein Rechteck $S \subseteq D$. Finde alle Datensätze (R,OId) in der Datei $\mathcal{R}$ mit $S \cap R \neq \emptyset$.

3. Rechteck-Umgebungs-Anfrage:
   Gegeben sei ein Rechteck $S \subseteq D$. Finde alle Datensätze (R,OId) in der Datei $\mathcal{R}$ mit $R \supseteq S$.

4. Rechteck-Inhalts-Anfrage:
   Gegeben sei ein Rechteck $S \subseteq D$. Finde alle Datensätze (R,OId) in der Datei $\mathcal{R}$ mit $R \subseteq S$.

5. Rechteck-Bereichsanfrage:
   Gegeben seien zwei Rechteck S_L, S_U mit $S_L \subseteq S_U \subseteq D$. Finde alle Datensätze (R,OId) in der Datei $\mathcal{R}$ mit $S_L \subseteq R \subseteq S_U$.

Neben diesen Anfragen soll sowohl das Einfügen wie auch das Löschen von Datensätzen in der Datei $\mathcal{R}$ ohne irgendwelche Restriktionen möglich sein. Darüber hinaus soll entsprechend zu PZS die Span möglichst hoch und die RZS dynamisch sein, d. h. Einfügen und Löschen von Datensätzen sollen nicht wesentlich die Leistung der RZS beeinflußen.

Neben diesen Effizienz-Kriterien ist die Flexibilität einer RZS besonders wichtig. Insbesondere fordern wir von einer RZS, daß neben dem MUR eines Objekts noch weitere k atomare Merkmale als k-dimensionaler Schlüssel K $\in [0,1)^k$, $k \geq 0$, deklariert werden können. Als Schlüssel verstehen wir hierbei die Attribute, die die Sortierung und Ordnung der Datensätze beeinflußen. Somit sind die Datensätze der Datei R in der allgemeinsten Form durch ein vierdimensionales Feld

$$(R_i, K_i, OId_i, I_i)$$

gegeben, wobei $R_i = Mur(OId_i) \subseteq D$, $K_i \in [0,1)^k$ und I_i, $1 \leq i \leq n$, den Informationsteil eines Datensatzes bezeichnet. Als Beispiel für einen zusätzlichen Schlüssel kann bei der Abspeicherung von Polygonen die Anzahl der Eckpunkte des Polygons in Frage kommen. Auch sollte es möglich sein, den Objektidentifikator selbst als Schlüssel oder als Teil eines Schlüssel zu deklarieren. Der Einfachheit halber betrachten wir im folgenden nur Datensätze der Form (R,OId) mit R = Mur(OId), wobei die Datei $\mathcal{R}$ bzgl. der MUR organisiert wird. Nur in speziellen Situationen werden wir auf die Problematik zusätzlicher Schlüssel verweisen.

Wie wir bei PZS und insbesondere bei PLOP-Hashing gesehen haben, erweisen sich abhängige Nicht-Gleichverteilungen als besonders problematisch. Demzufolge fordern wir von einer RZS, daß die Verteilung der Daten keinen Einfluß auf die Leistung der Verfahren haben soll. Zusätzlich müssen wir bei RZS den Überlappungsgrad der Datei betrachten. Der Überlappungsgrad hat ähnlich zu der Verteilung der Raumobjekte einen wesentlichen Einfluß auf die Leistung einer RZS.

Definition 4
Sei eine Datei $\mathcal{R} = \{(R_1, OId_1), \ldots, (R_n, OId_n)\}$, $R_i = Mur(OId_i) \subseteq D$ *gegeben, so ist zu einem Punkt* $P \in D$ *der* Überlappungsgrad *durch*

$$\Omega(\mathcal{R}, P) := \mid \{S \mid P \in S, (S, OId) \in \mathcal{R}\} \mid$$

gegeben. Der (maximale) Überlappungsgrad $O_{\mathcal{R}}$ *der Datei* $\mathcal{R}$ *ist gegeben durch*

$$\Omega := \Omega_{\mathcal{R}} := \max_{P \in D} \Omega(\mathcal{R}, P)$$

Somit verstehen wir unter dem Überlappungsgrad eines Punktes die Anzahl der Antworten bei einer entsprechenden Punkt-Anfrage und dem Überlappungsgrad der Datei entsprechend die maximale Anzahl von Antworten, die wir bei einer beliebigen Punkt-Anfrage erhalten können. Der Wert von Ω hängt stark von der jeweiligen Anwendung ab. Betrachten wir hierzu zwei typische Anwendungen im Bereich der Kartographie, so ergibt sich bei der Abspeicherung von Höhenlinien ein sehr hoher Überlappungsgrad, während bei der Abspeicherung von Parzellengrenzen ein sehr niedriger Überlappungsgrad vorliegt.

Die Anzahl der Datensätze und die Ausdehnung der Rechtecke beeinflußen entscheidend den Überlappungsgrad der Datei. Insbesondere soll sich die Ausdehnung der Rechtecke nicht auf die Leistung einer RZS auswirken. Bei der Abspeicherung von Rechtecken mit kleinem Volumen, wie auch von Rechtecken mit großem Volumen sollte die Effizienz einer RZS gleich bleiben. Da Punkte als spezielle Klasse von Recktecken angesehen werden können, sollte eine effiziente RZS auch die Anforderungen für eine effiziente PZS erfüllen.

Eine andere wichtige Forderung ist, eine RZS durch eine einfache Erweiterung einer PZS zu erzeugen. Die Möglichkeit, daß in einem DBS zu benutzerdefinierten Datentypen eigens dafür entwickelte Indexstrukturen implementiert werden, erscheint uns nicht praktikabel. Vielmehr sollten mehrdimensionale PZS so allgemein gehalten sein, daß durch eine einfache Spezifikation die gewünschte RZS entsteht.

5.4 Techniken für die Entwicklung von Raumzugriffsstrukturen

In den folgenden Abschnitten werden wir Techniken vorstellen, die es erlauben eine beliebige PZS in eine RZS zu überführen, wobei diese RZS die MUR der Objekte organisiert. Wir möchten dabei aufzeigen, daß im Prinzip jede bisher vorgestellte RZS auf den Techniken *überlappende Regionen, Transformationen* und *Clipping* basiert. Wir stellen dabei für PLOP-Hashing konkret die so entwickelten RZS vor. Schließlich stellen wir eine Hybrid-RZS basierend auf PLOP-Hashing vor, die alle obengenannten Techniken benutzt. Je nach Art der Raumdaten werden zur Laufzeit bei dieser Hybridstruktur (Hybrid-PLOP) die einzelnen Techniken aus Gründen der Effizienz gewichtet.

5.4.1 Clipping

Die Technik des Clipping kann am besten durch Beschreibung des Einfügens eines Rechtecks veranschaulicht werden. Nehmen wir dazu an, daß in eine durch PLOP-Hashing organisierte Datei bereits 2-dimensionale Rechtecke eingefügt wurden. Üblicherweise gibt es keinen Unterschied zwischen dem Einfügen eines 2-dimensionalen Rechtecks und dem eines 2-dimensionalen Punktes, falls das Rechteck ganz in einer Seitenregion liegt. Läßt sich ein Rechteck R nicht eindeutig zu einer Seite zuordnen (d. h. das Rechteck schneidet mindestens zwei Seitenregionen), so wird das Rechteck R in eine minimale Menge R^1,..., R^q von disjunkten Rechtecken zerlegt, so daß folgende Bedingungen erfüllt sind:

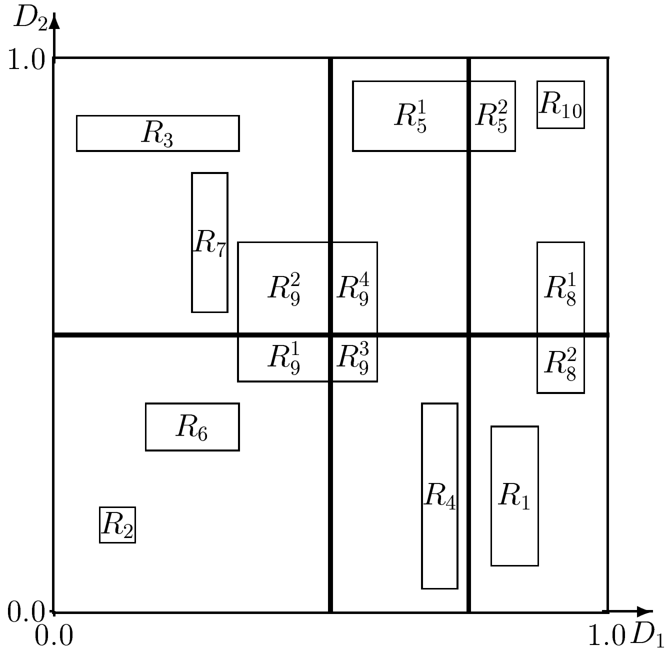


Abbildung 29: Partitionierung des Datenraums für eine Datei, die durch PLOP-Hashing und Clipping organisiert wird

1. $R = \cup_{i=1}^{q} R^i$
2. R^i wird von genau einer Seitenregion umschlossen, $1 \leq i \leq q$.

Die so entstandenen q Rechtecke können entsprechend den Punktdaten in die Datei eingefügt werden. Dabei wird mit Hilfe des Rechtecks R^i die Position bestimmt, wo das Rechteck eingefügt werden soll. Statt des zerschnittenen Rechtecks R^i, $1 \leq i \leq q$, wird das ursprüngliche Rechteck R auf diese Position geschrieben. RZS, die auf dem Prinzip von Clipping basieren, werden wir im folgenden als C-Verfahren bezeichnen.

In Abb. 29 haben wir die Partitionierung des Datenraums aufgezeigt nachdem zehn 2-dimensionale Rechtecke $R_1, \ldots, R_{10}$ eingefügt worden sind. Beispielhaft haben wir dabei PLOP-Hashing als PZS zugrunde gelegt. Die Datei besteht hierbei aus 6 Ketten, 3 Scheiben in der 1. Achse und 2 Scheiben in der 2. Achse. Die Rechtecke R_5 und R_9 sind in 2 bzw. in 4 Rechtecke zerlegt worden.

Der Vorteil dieser Methode besteht in ihrer Einfachheit. Zudem sind C-Verfahren eine echte Verallgemeinerung der zugrunde liegenden PZS, d. h. diese Verfahren sind sowohl für Punktdaten als auch für Raumdaten geeignet. Der offensichtliche Nachteil dieser Methode besteht in der Duplikation von Rechtecken und der damit verbundenen niedrigen Speicherplatzausnutzung. Zusätzlich ergibt sich ein erhöhter Aufwand für das Einfügen und Löschen von Rechtecken. So muß z. B. Rechteck R_9 viermal in die Datei eingefügt werden. Ein weiterer Nachteil dieser Verfahren ergibt sich aus folgender Beobachtung:

Sei $b > 1$ die Kapazität einer Datenseite und $\mathcal{R} = \{R_1, \ldots, R_n\}$ eine Menge von 2-dimensionalen Rechtecken, $n > b$. Verwenden wir zur Organisation der Rechtecke Clipping angewandt auf eine PZS ohne Überlaufsätze, so muß stets die Bedingung $\Omega_{\mathcal{R}} \leq b$ erfüllt sein.

Für Verfahren wie PLOP-Hashing, die Überlaufsätze zulassen, ist die Funktionalität der zugehörigen RZS garantiert. Trotzdem wird die Leistung solcher RZS eingeschränkt, falls die Bedingung $\Omega_{\mathcal{R}} \leq b$ verletzt ist. In solch einer Situation gibt es Rechtecke, die stets in der Sekundärdatei abgelegt werden und nicht durch Splitten der zugehörigen Datenkette in die Primärdatei umgespeichert werden können.

Für eine PZS, die Überlaufsätze vermeidet und die für die Realisierung einer RZS basierend auf Clipping verwendet wird, muß nun zusätzlich eine Überlaufbehandlung implementiert werden. In [MT 83] wurde dies z. B. für mehrdimensionales erweiterbares Hashing vorgestellt.

Betrachten wir Raum-Anfragen für RZS, die auf Clipping basieren. Hierbei setzen wir stets die Bedingung $\Omega_{\mathcal{R}} \leq b$ voraus. Diese RZS sind im gewissen Sinn optimal für die Beantwortung von Punkt- und Rechteck-Umgebungs-Anfragen. Für beide Anfragen genügt es, auf eine Datenseite, bzw. auf eine Kette von Datenseiten zuzugreifen. Dagegen muß bei der Beantwortung von Rechteck-Schnitt-Anfragen und Rechteck-Inhalts-Anfragen, unter Voraussetzung eines gemeinsamen Anfragerechtecks, auf dieselben Datenseiten zugegriffen werden. Bei einer Rechteck-Inhalt-Anfrage ist aber im Vergleich zu einer Rechteck-Schnitt-Anfrage i. a. mit wesentlich weniger Antworten zu rechnen. Ein weiterer Nachteil von C-Verfahren ergibt sich bei der Beantwortung von Rechteck-Schnitt- und Rechteck-Inhalts-Anfragen. Hierbei kann mehrmals auf geclippte Rechtecke zugegriffen werden, die zu einem Raumobjekt gehören. Somit kann unter der Voraussetzung, daß viele Rechtecke geclippt sind, für große Anfragebereiche die Leistung degenerieren.

Die Technik des Clippings kann auf eine beliebige PZS angewendet werden. So ist die zu Grunde liegende PZS des R^+-Baums [SRF 87] der k-d-B-Baum [Rob 81] und die des Box-Excell mehrdimensionales erweiterbares Hashing [Tam 82, Oto 84]. Alle diese C-Verfahren basieren also auf PZS, die Überlaufsätze vermeiden. Deshalb ist die Funktionalität der entsprechenden RZS eingeschränkt auf Anwendungen, in denen Bedingung $\Omega_{\mathcal{R}} \leq b$ erfüllt ist.

5.4.2 Überlappende Regionen

Entsprechend den C-Verfahren organisieren RZS mit der Technik der überlappenden Regionen (ÜR-Verfahren) eine Menge d-dimensionaler Rechtecke mit Hilfe einer d-dimensionalen PZS. Zur Erläuterung des Prinzips gehen wir kurz auf den R-Baum [Gut 84] ein, eine der bekanntesten RZS. Im folgenden verstehen wir unter einer Seitenregion, das MUR der Rechtecke, die zu dieser Seite gehören. Die Idee dieser Verfahren ist es, im Gegensatz zu C-Verfahren überlappende Seitenregionen zuzulassen.

Der R-Baum ist ein balancierter Baum, der den B^+-Baum für Raumobjekte verallgemeinert. Wir unterscheiden beim R-Baum zwischen Daten-Rechtecken, die den MUR der Objekte entsprechen und Directory-Rechtecken. Die Daten-Rechtecke werden stets in den Blättern des R-Baums gehalten, während sich Directory-Rechtecke in den inneren Knoten des R-Baums befinden. Zu jedem Directory-Rechteck gibt es einen Zeiger p, der auf einen Sohn verweist. Dabei ist das Directory-Rechteck das MUR der Rechtecke, die sich im

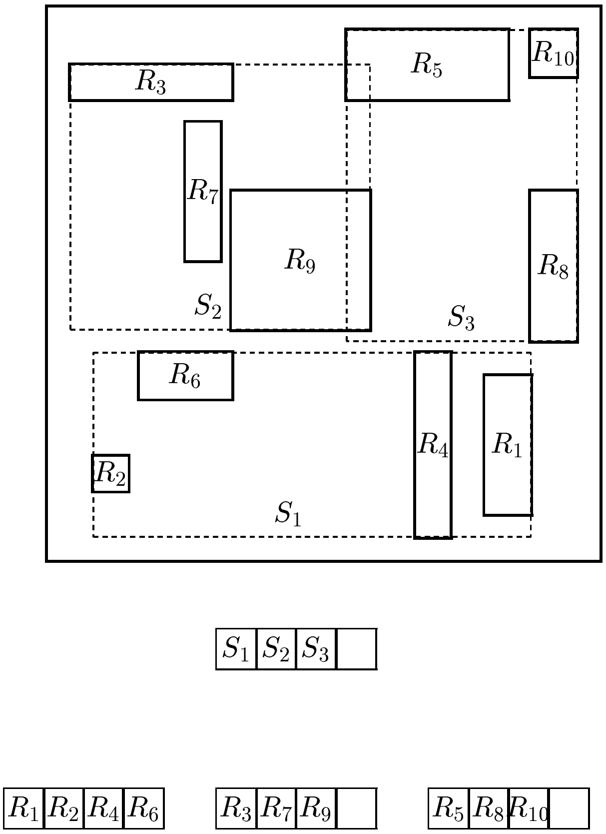


Abbildung 30: Organisation von zehn Rechtecken $R_1, \ldots, R_{10}$ durch den R-Baum

zugehörigen Sohn befinden. Entsprechend zum B^+-Baum, wächst der R-Baum von unten nach oben, so daß Erweiterungen des Baums auf einen Pfad von der Wurzel zu einem Blatt beschränkt bleiben. Dadurch ergibt sich auch, daß alle Blätter auf dem gleichen Level des Baums liegen. Da Clipping vermieden wird, muß ein Daten-Rechteck genau in einer Datenseite abgelegt werden. Somit können sich die Regionen der Datenseiten überlappen. In Figur 30 haben wir die Struktur eines R-Baums mit der entsprechenden Partitionierung des Datenraums veranschaulicht. Hierbei haben z. B. die Directory-Rechtecke S_2 und S_3 einen nicht-leeren Durchschnitt.

Ein Vorteil einer RZS, die die Technik der überlappenden Regionen verwendet, liegt in der hohen Speicherplatzausnutzung. So garantiert der R-Baum analog zu dem B^+-Baum eine Span von mindestens 50% und eine durchschnittliche Span von 69%. Ähnlich zu C-Verfahren können durch solch ein Verfahren sowohl d-dimensionale Punkte als auch d-dimensionale Rechtecke zusammen in einer Datei organisiert werden. Die Leistung des R-Baums [Gut 84], wie er von Guttman vorgestellt wurde, ist sehr stark von der Eingabenreihenfolge der Daten abhängig. Um diesen Nachteil zu beheben, sind speziell für den R-Baum sogenannte Pack-Algorithmen vorgestellt worden [RL 85], die in einem statischen Zustand der Datei die Leistung durch eine globale Reorganisation verbessern. Der beim Packen erzielte Leistungsgewinn kann aber bereits durch wenige Einfügungen wieder erheblich reduziert werden.

Weitere Varianten des R-Baums wurden in [Gre 89] und [BKSS 90] vorgestellt, wobei diese sich gegenüber der ursprünglichen Variante im wesentlichen durch einen anderen Splitalgorithmus unterscheiden. In einem experimentellen Leistungsvergleich [BKSS 90] hat sich dabei gezeigt, daß der R^*-Baum die robusteste und bei weitem effizienteste Variante des R-Baums ist.

Der R-Baum benötigt für eine Punkt-Anfrage teilweise mehr als einen Zugriff auf eine Datenseite und weitere Zugriffe auf Directory-Seiten. Sei in unserem Beispiel in Abb. 30 ein Punkt P mit $P \in S_2 \cap S_3$ der Suchpunkt einer Punkt-Anfrage, so muß auf zwei Datenseiten zugegriffen werden. Zudem werden, entsprechend zu C-Verfahren, bei Rechteck-Schnitt- und Rechteck-Umgebungs-Anfragen dieselben Datenseiten angesprochen. Prinzipiell hängt die Leistung sehr stark von der Größe und dem Überlappungsgrad der Rechtecke ab. Werden z. B. in eine Datei einige Rechtecke mit hoher Ausdehnung eingefügt, so wird der Überlappungsgrad der Daten-Rechtecke und dementsprechend der Directory-Rechtecke sehr stark anwachsen. Die unmittelbare Folgen sind ein hoher Verzweigungsgrad und hohe Anfragekosten. Wie in [FSR 87] gezeigt wurde, können dann C-Verfahren, wie der R^+-Baum, dem R-Baum,speziell bei Punkt-Anfragen wesentlich überlegen sein. Im Gegensatz dazu wird bei einer Rechteck-Schnitt-Anfrage der R-Baum dem R^+-Baum um so mehr überlegen sein, desto größer die Anfragebereiche werden.

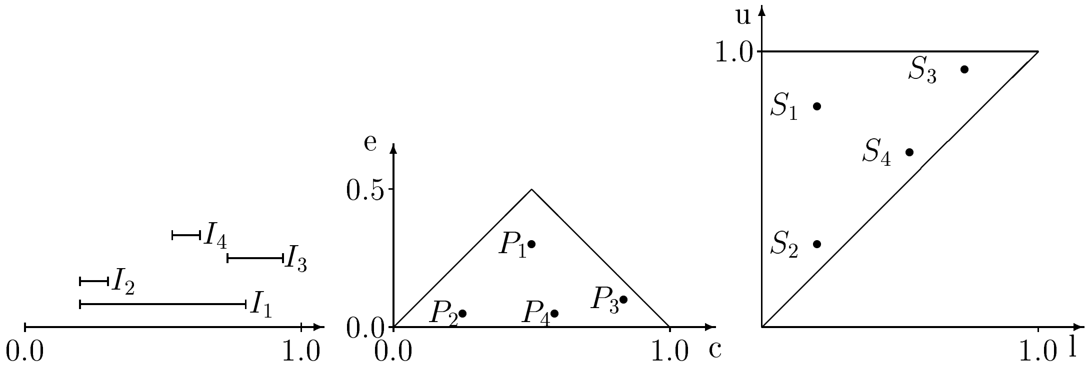


Abbildung 31: Transformation von Intervallen I_j in 2-dimensionale Punkte $P_j = (c_j, e_j)$ (Zentrums-Repräsentation) and $S_j = (l_j, u_j)$ (Ecken-Repräsentation), j = 1,..,4

Abschließend möchten wir erwähnen, daß in [Ooi 87] die Technik der überlappenden Regionen für den k-d-Baum vorgeschlagen wurde. Anhand von PLOP-Hashing werden wir in Abschnitt 5.5 aufzeigen, daß auch mehrdimensionale dynamische Hashverfahren sich als PZS für ÜR-Verfahren eignen. Der wesentliche Vorteil, als Basis für ÜR-Verfahren eine mehrdimensionale PZS zu nehmen (und nicht wie beim R-Baum eine eindimensionale PZS), liegt in der eindeutigen Zuordnung der MUR zu den Seiten. Insbesondere reduziert sich dadurch der Aufwand beim Einfügen und Löschen.

5.4.3 Transformation

Die wesentliche Idee von RZS, die auf der Technik von Transformation (T-Verfahren) basieren, ist die Repräsentation von d-dimensionalen Rechtecken durch 2d-dimensionale Punkte. So kann z. B. ein 2-dimensionales Rechteck, dessen Seiten parallel zu den Achsen des Datenraums liegen, durch einen 4-dimensionalen Punkt

$$(c_1, c_2, e_1, e_2)$$

repräsentiert werden (Zentrums-Repräsentation), wobei $c = (c_1,\ c_2) \in [0,1)^2$ der Schwerpunkt des Rechtecks ist und $e = (e_1,\ e_2) \in [0, 0.5)^2$ der Distanz des Schwerpunktes zu dem Rand entspricht. Wie in [NH 85] aufgezeigt, können diese Punkte durch die Gitterdatei oder ganz allgemein durch jede beliebige mehrdimensionale PZS organisiert werden.

Entsprechend zu ÜR-Verfahren bieten diese Verfahren den Vorteil, daß Objekte genau einmal in der Datei repräsentiert werden. Insbesondere werden bzgl. der Funktionalität keine weiteren Bedingungen an T-Verfahren gestellt.

Offensichtlich kann ein Rechteck auch durch den linken unteren $ll \in [0,1)^d$ und rechten oberen Eckpunkt $ru \in [0,1)^d$ repräsentiert werden (Ecken-Repräsentation). Die Wahl der jeweiligen Repräsentationsform kann aber die Leistung einer RZS wesentlich beeinflußen.

Der grundsätzliche Vorteil der Zentrums-Repräsentation ist, daß Informationen über die Lage des Rechtecks getrennt sind von der Information über die Ausdehnung. Darüber hinaus erscheint das Zentrum des Rechtecks unter allen Punkten, die im Rechteck liegen, die beste Näherung für die Position des Rechtecks zu sein.

Um noch eine Veranschaulichung der Methoden zu gewährleisten, beschränken wir uns im folgenden auf d=1. Ein 1-dimensionales Rechteck bezeichnen wir hierbei als Intervall. Dabei können wir ein Intervall $I \subseteq [0,1)$ durch die Zentrumsrepräsentation $((c,e),\ c \in [0,1),\ e \in [0,0.5))$, und zum anderen durch die Ecken-Repräsentation $((l,u), 0 \leq l \leq u < 1)$ darstellen.

Ein weiterer Vorteil der Zentrums-Repräsentation liegt darin, daß PZS wie z. B. die Gitterdatei oder PLOP-Hashing einen rechteckigen Datenraum voraussetzen. Die Leistung dieser RZS verschlechtert sich genau dann, wenn es große Bereiche des Datenraums gibt, in denen keine Daten liegen. Im Fall der Eckenrepräsentation ist der Datenraum das Einheitsquadrat, wobei nur oberhalb der Diagonalen Daten liegen, d. h. nur in einer Hälfte des Datenraums liegen Datensätze, siehe Abb. 31. Betrachten wir 3-dimensionale Rechtecke, so können nur noch in einem 1/8 des 6-dimensionalen transformierten rechteckigen Datenraums $[0,1)^6$ Daten liegen. Für die Zentrums-Repräsentation ergibt sich augenscheinlich eine entsprechende Situation, siehe Abb. 31. Da aber in den meisten Anwendungen die Volumen der Objekte (und damit auch deren MUR) relativ klein zum geometrischen Datenraum sind, kann man den Datenraum zu einem Trapez

$$\{(c,e) \mid 0 \leq e \leq \min\{emax, c, 1.0 - c\}\}$$

verkleinern, wobei emax, $emax \in [0,0.5)$, die maximale Länge eines in der Datei vorkommenden Intervalls ist. Für $emax \approx 0$ erhalten wir also annähernd einen rechteckigen Datenraum. Dagegen liegen bei der Ecken-Repräsentation die transformierten Punkte nahe bei der Diagonalen des Datenraums. Diese Verteilung entspricht in etwa dem schlechtesten Fall der Gitterdatei, in welchem das Directory in $O(n^d)$ anwächst.

Für PZS, wie das BANG-File [Fre 87] oder den Buddy-Hashbaum, dürften aber obige Überlegungen keine wesentliche Rolle spielen. Wie in Abschnitt 3 beschrieben, organisiert der Buddy-Hashbaum nur die Teile des Datenraums, in welchen auch Datensätze liegen. Leere Bereiche des Datenraums werden im Buddy-Hashbaum nicht repräsentiert. Für das BANG-File könnte sich die Eigenschaft, daß die Partitionierung des Datenraums vollständig ist, negativ auf die Leistung auswirken.

Die verschiedenen 2-dimensionalen Punkt-Repräsentationsformen von Intervallen lassen sich durch eine einfache *lineare Abbildung* (Drehung und Translation) ineinander überführen:

$$\begin{aligned} c &= u/2 + l/2 \\ e &= u/2 - l/2 \end{aligned} \tag{7}$$

Da Intervalle durch Transformation in 2-dimensionale Punkte überführt wurden, müssen entsprechend die Anfragebereiche dieser Transformation unterzogen werden. Für die Veranschaulichung der transformierten Suchbereiche betrachten wir zunächst die Ecken-Repräsentation der Intervalle. Wir erhalten für die Anfragen (1) - (4) rechteckige Suchbereiche. Beispielsweise wird eine Punkt-Anfrage in unserem transformierten Datenraum zu einer Bereichsanfrage, wobei der rechte untere Punkt auf der Diagonalen liegt und der linke obere Punkt dem linken oberen Eckpunkt des Datenraums entspricht.

Da wir durch die Gleichungen 7 die Ecken-Repräsentation in eine Zentrums-Repräsentation überführen können, erhalten wir die entsprechenden Suchregionen für die Zentrums-Repräsentation, indem wir die Suchregionen der Ecken-Repräsentation mit Hilfe dieser

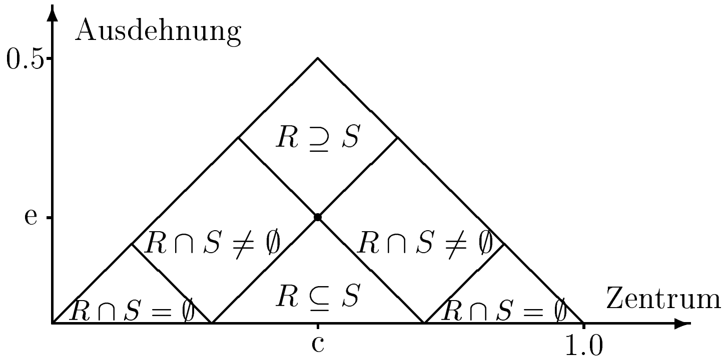


Abbildung 32: Für die Zentrums-Repräsentation und einem vorgegebenen Intervall S=(c,e), sind die Bereiche des Datenraums dargestellt, in welchen alle Intervalle R mit $R \supseteq S$,$R \subseteq S$ und $R \cap S \neq \emptyset$ liegen

Gleichungen in die Suchregionen der Zentrums-Repräsentation überführen. Die Form der einzelnen Suchregionen entspricht dann der eines Kegels, siehe Abb. 32.

Der Vorteil der Transformation d-dimensionaler Rechtecke in 2d-dimensionale Punkte, ist die klare Trennung der einzelnen Suchregionen. Gegenüber Verfahren, die auf der Technik der überlappenden Regionen oder Clipping basieren, muß bei T-Verfahren i. a. bei einer Rechteck-Schnitt-Anfrage und einer Rechteck-Inhalt-Anfragen nicht auf dieselben Seiten zugegriffen werden. Durch diese klare Trennung der einzelnen Suchregionen ist zunächst mit einer verbesserten Leistung gegenüber C-Verfahren zu rechnen.

Dieser Vorteil gegenüber den auf anderen Techniken basierenden RZS wird aber dadurch reduziert, daß der Rand der Suchregion eine höhere Dimension besitzt. Dadurch müssen T-Verfahren auf viele Seiten zugreifen, in denen Intervalle liegen, die nicht die Anfrage beantworten. Für d=1 kann bei einem C-Verfahren z. B. der Rand einer Intervall-Schnitt-Anfrage maximal aus zwei Datenseiten bestehen, während der Rand für ein T-Verfahren unter Voraussetzung einer symmetrischen Partitionierung des Datenraums O($\sqrt{n}$) Seiten schneidet. Indirekt hängt dieser Nachteil der T-Verfahren damit zusammen, daß nah beieinanderliegende Intervalle im transformierten Datenraum weit auseinander liegen können. Durch eine entsprechende Partitionierung des Datenraums kann aber dafür gesorgt werden, daß diese Intervalle trotzdem physisch nah zusammen abgespeichert werden. In Abschnitt 5.6 werden wir ausführlich auf dieses Problem eingehen.

Aus den Betrachtungen in den letzten drei Abschnitten können wir folgendes Resümee ziehen:

- C-Verfahren müssen auf PZS basieren, die Überlaufsätze erlauben, da sonst die Funktionalität der RZS bei einem hohen Überlappungsgrad eingeschränkt ist. C-Verfahren sind gut für Punkt und Rechteck-Umgebungs-Anfragen geeignet, da hierfür genau ein Zugriff auf eine Datenseite, bzw. auf eine Kette von Datenseiten notwendig ist. Bedingt durch das mehrfache Abspeichern der Rechtecke, reduziert sich die Leistung bei Rechteck-Schnitt-Anfragen um so mehr, desto größer der zugehörige Anfragebereich ist.

- ÜR-Verfahren und T-Verfahren repräsentieren ein Objekt durch ein Rechteck und besitzen somit geringe Einfügekosten sowie eine hohe Span.
- ÜR-Verfahren haben den wesentlichen Nachteil, daß beim Einfügen von volumengroßen Rechtecken die Effizienz der Verfahren stark beeinträchtigt wird.
- Bei T-Verfahren besteht das Problem in der Declusterung von nah beieinander liegenden d-dimensionalen Rechtecken in weit auseinander liegenden 2d-dimensionale Punkte. Zudem besitzt der Rand einer Anfrageregion eine hohe Dimension, so daß auf viele Datensätze zugegriffen wird, die die Anfrage nicht erfüllen.

In den nächsten Abschnitten werden wir ein Verfahren vorstellen, das sowohl die Technik der überlappenden Regionen als auch der Transformation gemeinsam benutzt. Dieses Hybrid-RZS basiert auf PLOP-Hashing und nutzt dessen spezifische Eigenschaften sehr gut aus. Weiterhin werden wir eine neue Variante des Clipping vorschlagen, die wir als *kontrolliertes Clipping* bezeichnen. Im Prinzip ist kontrolliertes Clipping auf jede RZS anwendbar. Es bewirkt bei ÜR-Verfahren und T-Verfahren eine verbesserte Leistung bei Raum-Anfragen zulasten der Span.

5.5 Überlappende Regionen angewandt auf PLOP-Hashing

In diesem Abschnitt zeigen wir auf, daß die Technik der ÜR nicht nur auf Baumverfahren anwendbar ist, sondern daß MDH wie PLOP-Hashing in Kombination mit dieser Technik ebenfalls eine effiziente RZS ergeben. Im folgenden betrachten wir eine Datei $\mathcal{R} = \{(R_1, OId_1), \ldots, (R_n, OId_n)\}$, wobei OId_i ein 2-dimensionales Object O_i identifiziert und $R_i = Mur(OId_i)$, $1 \leq i \leq n$, das zugehörige 2-dimensionale MUR bezeichnet. Ein Rechteck R mit $(R, OId) \in \mathcal{R}$ ist dabei durch einen 4-dimensionalen Punkt (c_1, c_2, e_1, e_2) in Zentrumsrepräsentation gegeben, $(c_1, c_2) \in [0,1)^2$, $(e_1, e_2) \in [0, 0.5)^2$.

Wie in Abschnitt 2 angesprochen, partitioniert PLOP-Hashing den Datenraum mit Hilfe eines orthogonalen Gitters. Dieses Gitter wird durch d binäre Bäume spezifiziert, die im Hauptspeicher gehalten werden. Wie in [Ooi 87] für den k-d-Baum vorgeschlagen, soll allein die Zentrumskomponente eines Rechtecks R die Position des zugehörigen Datensatzes (R,OId) in der Datei $\mathcal{R}$ beeinflußen. Wir repräsentieren somit ein d-dimensionales Rechteck durch den zugehörigen d-dimensionalen Schwerpunkt.

Ergänzend zu der bisherigen Struktur von PLOP-Hashing speichern wir in einem Blatt mit Index i_j des j-ten binären Baums, $0 \leq i_j \leq m_j$, $1 \leq j \leq d$, (d. h. das Blatt gehört zu der Scheibe S(i_j, j)) folgende Werte ab:

$$min(i_j, j) \;:=\; \min\{c - e \mid c \in S(i_j, j), R = (c, e), (R, OId) \in \mathcal{R}\} \qquad (8)$$

$$max(i_j, j) \;:=\; \max\{c + e \mid c \in S(i_j, j), R = (c, e), (R, OId) \in \mathcal{R}\} \qquad (9)$$

Der Parameter min(i_j, j), bzw. max(i_j, j) entspricht somit dem am weitesten links, bzw. rechts liegenden Eckpunkt eines Rechtecks, das zu der Scheibe S(i_j, j) gehört. Diese Information wird benötigt, um Kenntnis über die Ausdehnung der in einer Scheibe liegenden Rechtecke zu bekommen. Auf diese Werte wird explizit bei entsprechenden Anfragen zugegriffen. In Abb. 33 haben wir die Partitionierung der Datei und die binären Bäume von PLOP-Hashing aufgezeigt, nachdem 10 Rechtecke $R_1, \ldots, R_{10}$ in die Datei eingefügt

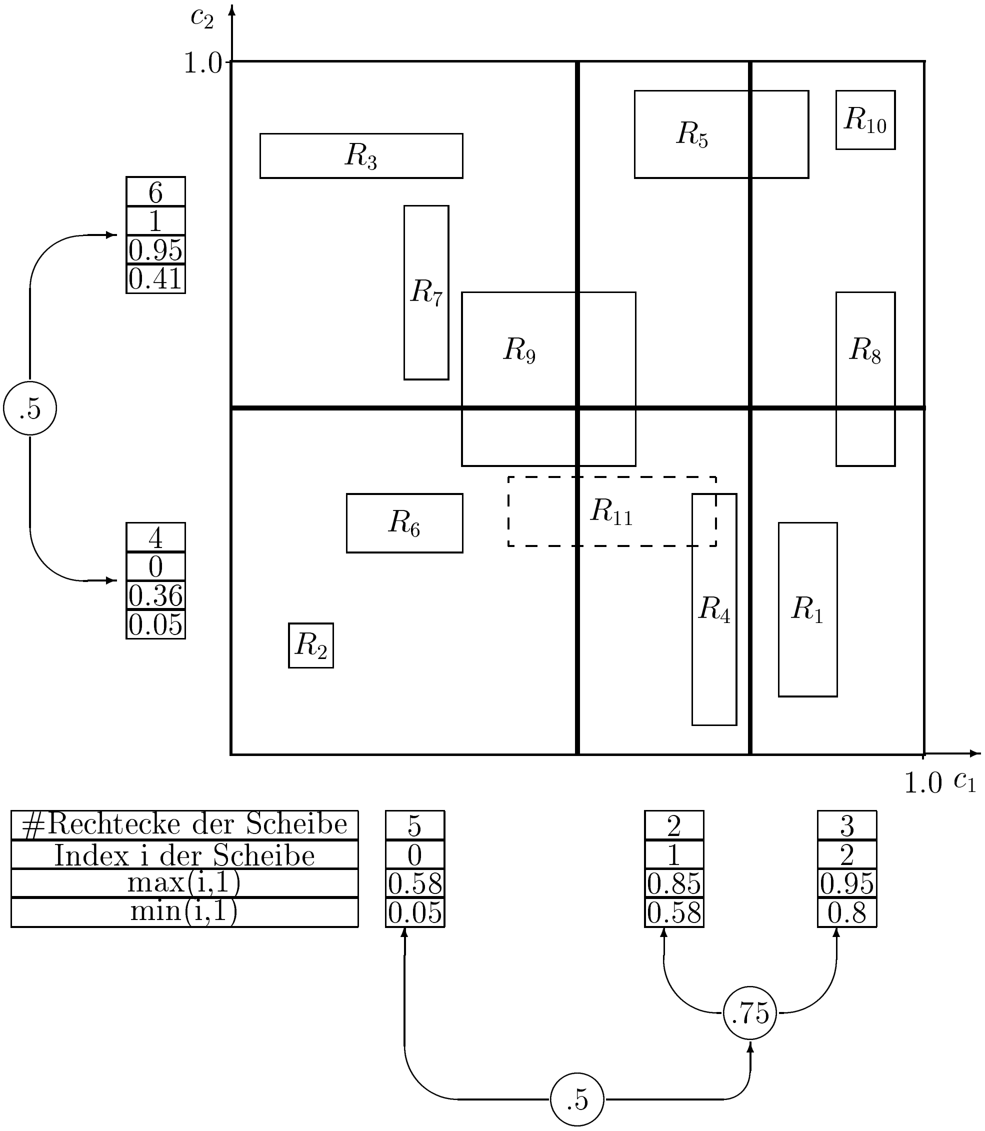


Abbildung 33: PLOP-Hashing in Kombination mit der Technik von überlappenden Regionen

wurden. Obwohl das Rechteck R_9 4 Gitterzellen schneidet, wird es mit dem Rechteck R_3 gemeinsam in einer Seite abgespeichert, da die Zentren dieser Rechtecke gemeinsam in der linken oberen Zelle liegen. Bei der exakten Suche besteht kein Unterschied zu dem Suchalgorithmus PLOP_EMQ des PLOP-Hashings, siehe Abschnitt 2.4. Als Eingabe für eine exakte Suche eines Rechtecks benötigen wir hierbei nur die Zentrumskoordinaten. Das Einfügen eines Datensatzes wird durch den Algorithmus InsertRectangle wie folgt realisiert.

Algorithmus InsertRectangle
gegeben: eine durch PLOP-Hashing organisierte Datei $\mathcal{R} = \{(R_1, OId_1), \ldots, (R_n, OId_n)\}$, ein Datensatz (R,OId), $R = ((c_1, \ldots, c_d), (e_1, \ldots, e_d))$.

```
1. Führe eine exakte Anfrage PLOP_EMQ($c_1, \ldots, c_d$) nach dem Datensatz (R,OId)
   durch;
   (* Dabei sei $\vec{i} = (i_1, \ldots, i_d)$ das Indexfeld, das in Schritt 3 des Algorithmus PLOP_EMQ
   zur Bestimmung der Adresse benutzt wurde *)

2. IF (R,OId) $\in \mathcal{R}$ THEN
     WriteErrorMsg("Datensatz existiert bereits in der Datei"); EXIT
   ELSE
     füge den Datensatz in die entsprechende Seite ein
   END;

3. (* Anpassung der min- und max-Werte *)
   FOR $j := 1$ TO d DO
       IF $(c_j - e_j) < min(i_j, j)$ THEN
           min $(i_j, j) := c_j - e_j$
       END;
       IF $c_j + e_j > max(i_j, j)$ THEN
           max $(i_j, j) := c_j + e_j$
       END;
   END;
```

Im Gegensatz zum R-Baum haben wir eine eindeutige Zuordnung von Rechtecken zu Datenseiten, da als Basis eine mehrdimensionale PZS verwendet wird und die Position, an welcher das Rechteck abgespeichert wird, eindeutig durch das Zentrum gegeben ist. Insbesondere wird dadurch das Einfügen und Löschen von Datensätzen effizienter ausgeführt. Fügen wir in unserem Beispiel in Abb. 33 den Datensatz (R_{11}, OId_{11}) mit $c_{11} = (0.55, 0.35)$ und $e_{11} = (0.15, 0.05)$ in die Datei ein, so arbeitet der Algorithmus InsertRectangle wie folgt. Zunächst wird durch PLOP_EMQ(0.55, 0.35) die Kette von Seiten bestimmt, in welche der zugehörige Datensatz eingefügt wird. Zusätzlich bekommen wir das zu dieser Kette gehörende Indexfeld $\vec{i} = (1, 0)$ geliefert. Nachdem wir in Schritt 2 den Datensatz in die entsprechende Kette eingefügt haben, müssen wir in Schritt 3 überprüfen, ob die Werte min(1,1), max(1,1), min(0,2) und max(0,2) neu angepaßt werden müssen. Hierbei wird min(1,1) := 0.4 und max(0,2) := 0.4 neu berechnet.

Durch das Löschen von Datensätzen können sich ebenfalls die min- und max-Werte der Scheiben ändern. Eine Anpassung dieser Werte würde nach dem Löschen im schlechtesten

Fall das Durchsuchen von d Scheiben erfordern. Um diese hohen Kosten zu vermeiden, werden diese Werte nicht direkt nach dem Löschen, sondern erst bei der nächsten Split- oder Verschmelze-Operation der jeweiligen Scheibe angepaßt.

Zur Erklärung der erweiterten Struktur der binären Bäume stellen wir den Algorithmus für eine Punkt-Anfrage vor. Hierfür nehmen wir an, daß die Datei weder erweitert noch verkleinert wird.

Algorithmus PointQuery

gegeben: eine durch PLOP-Hashing organisierte Datei $\mathcal{R} = \{(R_1, OId_1), \ldots, (R_n, OId_n)\}$, Punkt K $= (K_1, \ldots, K_d) \in D$ und emax eine obere Schranke für die maximale Ausdehnung der Rechtecke

```
1. FOR j := 1 TO d DO
       $low_j$ := Maximum(0, $K_j - emax$); $up_j$ := Minimum(1,$K_j + emax$);
   END;

2. (* Erzeugung von Indexmengen $I_j,\ 1 \le j \le d$ *)

   FOR j := 1 TO d DO
       $I_j := \{\}$;
       fromleaf := Traverse($B_j, low_j$);
       toleaf := Traverse($B_j, up_j$);
       LOOP
           i := fromleaf↑.index;
           IF ( min(i,j) $\le K_j \le$ max(i,j) ) THEN
               $I_j := I_j \cup \{i\}$
           END;
           IF toleaf = fromleaf THEN EXIT END;
           fromleaf := fromleaf↑.right;
       END;
   END;

3. (* Bildung des kartesischen Produkts über die Indexmengen $I_1, \ldots, I_d$ *)

   FOR $\vec{i} \in I_1 \times \ldots, I_d$ DO
       adr := DLA($\vec{i}, \vec{m}$);
       (* DLA ist die Adreßfunktion von PLOP-Hashing, siehe Abschnitt 2.2 *)
       Durchsuche die Kette mit Adresse adr nach möglichen Antworten
   END;
```

Der Algorithmus *PointQuery* arbeitet ähnlich dem *Range-Query*-Algorithmus des ursprünglichen PLOP-Hashings. Für $d = 2$ wird zunächst durch $[low_1, up_1] \times [low_2, ip_2]$ in Schritt 1 der maximal zu durchsuchende Bereich festgelegt. Dabei werden zwei Funktionen *Minimum* und *Maximum* benutzt, die jeweils das Minimum und Maximum von zwei Zahlen berechnen. Zu beachten ist, daß für genügend kleine Werte von emax nicht sämtliche Blätter der binären Bäume durchsucht werden müssen und dies somit zu einer Reduzierung der CPU-Zeit führt. Dieser Parameter hat aber *keine* Auswirkungen auf die Anzahl der Diskzugriffe. Im 2. Schritt werden dann in jeder Achse die Scheiben notiert, in denen möglicherweise ein Rechteck liegt, das den Punkt K enthält. Um dies effizient zu überprüfen, benötigen wir die zusätzlichen *min*- und *max*-Werte in den Blättern der binären Bäume. Schließlich werden in Schritt 3 die Adressen der zugehörigen Ketten berechnet und anschließend diese Ketten durchsucht.

Zur Veranschaulichung des Algorithmus PointQuery führen wir in unserem Beispiel aus Abb. 33 für emax = 0.2 und den Punkt K=(0.82,0.25) eine Punkt-Anfrage durch. Zu beachten ist, daß $K_1 = \{.5, .75\}$, $K_2 = \{.5\}$ und somit $\vec{m} = (2, 1)$ gilt. In Schritt 1 erhalten wir $low_1 = 0.62$, $low_2 = 0.05$, $up_1 = 1$, $up_2 = 0.45$. Im 2. Schritt müssen wir für j=1 die Scheiben S(1,1) und S(2,1) überprüfen, ob diese Antworten enthalten können. Da für beide Scheiben diese Möglichkeit besteht, erhalten wir $I_1 = \{1, 2\}$. Entsprechend müssen wir für die 2. Achse die Scheiben S(0,2) und S(1,2) überprüfen und wir erhalten $I_2 = \{0\}$. Schließlich durchsuchen wir im 3. Schritt die Ketten mit Adressen DLA((0,0),(2,1)) und DLA((1,0),(2,1)) nach möglichen Antworten.

Entsprechend zum R-Baum, wird i. a. bei einer Punkt-Anfrage auf mehr als eine Datenkette zugegriffen. Für Rechtecke mit geringer Ausdehnung, ist die Technik der überlappenden Regionen sehr effizient, da der Überlappungsgrad der Seitenregionen entsprechend niedrig ist. Durch Einfügung einiger Rechtecke mit hoher Ausdehnung kann die Leistung des Verfahrens sehr stark abfallen. Der Einfluß sehr großer Rechtecke auf die Leistung unseres Verfahrens, kann nur dadurch verhindert werden, daß große Rechtecke in kleinere Rechtecke zerschnitten werden (Clipping) oder daß große Rechtecke gesondert behandelt werden. Wir werden im folgenden auf den zweiten Ansatz näher eingehen.

Ein Vorschlag für eine gesonderte Behandlung von großen Rechtecken könnte sein, - ähnlich zu der Mehrschichten-Gitterdatei [SW 88] - mehrere Dateien gleichzeitig durch PLOP-Hashing zu organisieren, wobei die Rechtecke nach einem gewissen Kriterium diesen verschiedenen Dateien zugeteilt werden. Ein Kriterium hierfür könnte z. B. die Ausdehnung der einzelnen Rechtecke sein, d. h. in Abhängigkeit der Ausdehnung des Rechtecks wird es einer Datei zugeordnet. Unter der Annahme, daß viele Rechtecke mit geringer Ausdehnung und nur wenige mit einer hohen Ausdehnung in der Datei liegen, kann eine erhebliche Reduzierung des Überlappungsgrades und damit i. a. eine verbesserte Leistung bei Rechteck-Schnitt-Anfragen erreicht werden.

Ein Nachteil ergibt sich durch die Organisation von einer logischen Datei in mehreren physischen Dateien. Auch stellen sich die Fragen, wieviel physische Dateien benutzt werden sollen und ob die Anzahl der Dateien in Abhängigkeit des Überlappungsgrads sich dynamisch ändern kann. Unsere Idee ist es, falls Rechteckdaten mit der ÜR-Variante des PLOP-Hashings organisiert werden, gegebenenfalls die Dimension des Datenraums zu erhöhen. Genauer gesagt lassen wir die Ausdehnung neben dem Zentrum der Rechtecke als weiteres Sortierkriterium für die Rechteckdaten genau dann zu, wenn einige Rechtecke zu groß sind und somit ein hoher Überlappungsgrad in der Datei $\mathcal{R}$ vorliegt.

Dieser Ansatz sorgt dafür, daß Rechtecke kompakt in *einer* Datei organisiert werden, wobei die Anzahl der Partitionierungspunkte in den Erweiterungsachsen nicht durch Vorgaben beschränkt ist. Bemerkenswert an diesem Ansatz ist, daß die Technik der Überlappenden Regionen direkt in die Technik der Transformation übergeht, wobei die einzelnen Achsen des Datenraums nicht gleich behandelt werden. Dies wird insbesondere dadurch ermöglicht, daß beide Techniken das Zentrum eines Rechtecks als Kriterium zum Unterteilen des Datenraums verwenden. Unter Nicht-Gleichbehandlung der Achsen verstehen wir, daß das Verhältnis der Anzahl der Partitionierungspunkte in den einzelnen Achsen beliebig variieren kann. Wir möchten hier betonen, daß es nicht zwingend ist, nach den Ausdehnungsachsen den Datenraum zu unterteilen. Für diesen Fall benötigen wir die Technik der ÜR, um die Effizienz des Verfahrens bei entsprechenden Anfragen zu erhöhen. Wie das Kriterium für die Wahl der nächsten Splitachse aussehen soll und wann die Erweiterungsachsen als Sortierschlüssel hinzugenommen werden sollen, werden wir ausführlich in dem nächsten Abschnitt diskutieren.

5.6 Asymmetrisches Partitionieren des Datenraums

Wie im letzten Abschnitt aufgezeigt wurde, reduziert sich die Effizienz von ÜR-Verfahren, falls Rechtecke mit hoher Ausdehnung eingefügt werden. In unserem Vorschlag wird deshalb für PLOP-Hashing die Ausdehnung der Rechtecke als zusätzliche Achse des Datenraums eingeführt. Wir erhalten dadurch eine RZS, die auf der Technik der Transformation von Rechtecken basiert. Der wesentliche Unterschied zu [NH 85] besteht darin, daß die einzelnen Achsen des Datenraums nicht gleich behandelt werden. Insbesondere hängt die Anzahl der Partitionierungspunkte m_j in der j-ten Achse, $1 \leq j \leq d$, von der Verteilung und dem Überlappungsgrad der Rechtecke ab. In diesem Abschnitt entwickeln wir Kriterien, welche Achse nach einer Verdopplung der Datei zur nächsten Splitachse bestimmt werden soll. Ziel ist es dabei, die Anfragekosten zu minimieren. Dabei beschränken wir unsere Betrachtungen auf Rechteck-Schnitt- und Punkt-Anfragen.

Im folgenden setzen wir d=1 voraus, wobei wir die eindimensionalen Rechtecke als Intervalle bezeichnen. Weiterhin nehmen wir an, daß ein Intervall $I = (c, e) \subseteq [0, 1)$ in Zentrumsrepräsentation gegeben ist, $c \in [0, 1),\ e \in [0, 0.5)$. Wie bereits oben erwähnt, können wir den rechteckigen, zu partitionierenden Datenraum für die Zentrums-Repräsentation auf

$$T_{emax} = \{(c, e) \mid 0 \leq c < 1, 0 \leq e \leq emax\} \tag{10}$$

reduzieren, wobei emax die maximale Ausdehnung eines Intervalls in der Datei ist. Weiterhin nehmen wir an, daß die c-Achse stets der ersten und die e-Achse der zweiten Achse entspricht. Somit ergibt sich, daß P_1 der Menge der Partitionierungspunkte in der c-Achse entspricht, sowie $m_1 = \mid P_1 \mid$ und S(i,1), $0 \leq i \leq m_1$, die Scheiben in der c-Achse bezeichnen. Zusätzlich betrachten wir sogenannte *lokale Levels* $l_1 = \lfloor \log_2 m_1 \rfloor$ und $l_2 = \lfloor \log_2 m_2 \rfloor$, welche somit die Gleichung $L = l_1 + l_2$ erfüllen. Unabhängig davon, ob Datensätze gelöscht oder eingefügt werden, soll der Wert von emax genau bekannt sein. Dies kann nur in einfacher Weise bei PLOP-Hashing realisiert sein, wenn für jede Scheibe in der e-Achse folgende Werte bekannt sind:

$$emax_i = \{e \mid I = (c, e) \text{ ist ein Intervall in der Datei } \mathcal{R} \text{ mit}(c, e) \in S(i, 1)\} \tag{11}$$

Für eine stets anwachsende Datei benötigen wir diese Werte zur Betimmung von emax nicht. Dagegen ermöglichen nach dem Löschen des längsten Intervalls die Werte $emax_i$ eine gute Abschätzung für den Wert emax.

Um den Leistungsunterschied für PLOP-Hashing für eine asymmetrische Partitionierung gegenüber einer symmetrischen Partitionierung analytisch aufzuzeigen, beschränken wir unsere Betrachtungen auf eine Gleichverteilung der Daten. Unter Gleichverteilung verstehen wir hier, daß die 2-dimensionalen Punkte $(c, e) \in T_{emax}$ einer *unabhängigen* Gleichverteilung folgen. Es ist uns hierbei klar, daß diese Annahme nicht realistisch ist. Solche Annahmen ermöglichen aber erst einen analytischen Vergleich. Insbesondere können die Leistungsunterschiede der einzelnen Verfahren dadurch leicht veranschaulicht werden.

Theorem 3
Seien die Intervalle I=(c,e) gleichverteilt in T_{emax}. Weiterhin nehmen wir an, daß PLOP-Hashing eine Datei mit 2^L Ketten organisiert, wobei das zugrunde liegende Gitter G_D den Datenraum in gleichgroße Zellen aufteilt. Weiterhin bezeichnet l das lokale Level der 1. Achse. Dann ergibt sich die durchschnittliche Anzahl von Gitterzellen, die durch eine Intervall-Schnitt-Anfrage eines Intervalls $S \subseteq [emax, 1.0 - emax]$ mit fester Ausdehnung 2σ, $0 < \sigma < 0.5$, getroffen werden, durch:

$$A_L(l) := emax * (2^l + 2^L) + 2^{L-l} + 2\sigma * 2^L \tag{12}$$

Beweis:
Sei $\sigma \in [0, 0.5)$ fest vorgegeben, so daß für das Intervall S = (c,σ), $c \in [0, 1), S \subseteq [emax, 1.0 - emax]$ gilt. Zu dem Anfrageintervall S suchen wir die Anzahl aller Gitterzellen im 2-dimensionalen transformierten Datenraum, die den transformierten Anfragebereich schneiden. Dieser Bereich ist ein Viereck des Datenraums mit Flächeninhalt $emax^2 + 2emax * \sigma$, siehe Abb. 34. Zwei Seiten des Vierecks liegen parallel zu der c-Achse, während die anderen Seiten einen 45°, bzw. 135° Winkel mit der c-Achse bilden. Da wir

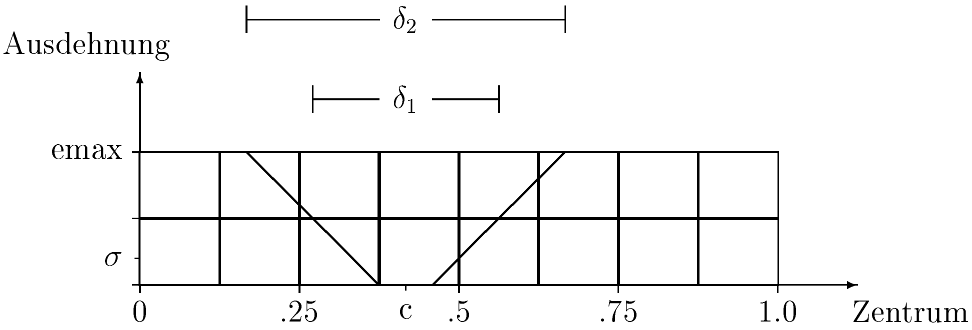


Abbildung 34: Partitionierung des PLOP-Hashings, wobei $L = 4$, $l_1 = 3$, $l_2 = 1$, $P_1 = \{0, 1/8, .., 7/8, 1\}$, $P_2 = \{0, emax/2, emax\}$

Gleichverteilung in T_{emax} voraussetzen und der Datenraum T_{emax} einen Flächeninhalt von emax besitzt, ergeben sich im Durchschnitt $(emax + 2\sigma) * n$ Antworten pro Intervall-Schnitt-Anfrage. Zur Ermittlung der Anzahl der im Suchbereich liegenden Zellen benötigen wir die Länge δ_i der Liniensegmente Q_i, $0 \leq i \leq 2^{l_2}$, die wir erhalten, indem wir den Suchbereich mit den Hyperebenen der e-Achse schneiden. Aufgrund der gleichmäßigen Partitionierung des Datenraums erhalten wir

$$\delta_i = 2i * emax/2^{l_2} + 2\sigma \;, \quad 0 \leq i \leq 2^{l_2}$$

Der Erwartungswert E_i, wieviele Hyperebenen der c-Achse das Liniensegment Q_i schneiden ergibt sich zu

$$E_i = \delta_i * 2^{l_1} = 2i * emax * 2^{l_1 - l_2} + 2\sigma * 2^{l_1} \;, \quad i = 0, \ldots, 2^{l_2}$$

Die erwartete Anzahl von Gitterzellen, die den Suchbereich schneiden ergibt sich aus

$$\sum_{i=1}^{2^{l_2}} (E_i + 1) = emax * (2^{l_1} + 2^L) + 2^{l_2} + 2\sigma * 2^L$$

Da $l_1 + l_2 = L$ gilt, erhalten wir die Gleichung 12 in Abhängigkeit des Levels der ersten Achse.

Folgerungen:

- Unter der Annahme der Gleichverteilung der Intervalle im Datenraum T_{emax} ist die erwartete Anzahl von Antworten für eine Intervall-Schnitt-Anfrage mit dem Anfrageintervall $S = (c, \sigma) \subseteq T_{emax}$ wie folgt gegeben:

  $$(emax + 2\sigma) * n$$

- Betrachten wir $A_L(l)$ als eine reellwertige Funktion. Der Wert l_{min} mit $A_L(l_{min}) = \min_{0 \leq l \leq L} A_L(l)$ kann mit Hilfe der Ableitung von $A_L(l)$ durch $A'_L(l_{min}) = 0$ berechnet werden. Wir erhalten

  $$l_{min} = \begin{cases} L & \text{falls } emax * 2^L < 1 \\ (L - \log_2 emax)/2 & \text{sonst} \end{cases} \tag{13}$$

  Der minimale Wert $A_{L,min} = A_L(l_{min})$ der Funktion $A_L(l)$ ergibt sich zu

  $$A_{L,min} = \begin{cases} 2(emax + \sigma) * 2^L + 1 & \text{falls } emax * 2^L < 1 \\ (emax + 2\sigma) * 2^L \; + \; 2 * (emax * 2^L)^{1/2} & \text{sonst} \end{cases} \tag{14}$$

Sei nun $2^L * emax \geq 1$ und $\widehat{l}_{min}$ der ganzzahlige Wert, an dem die Funktion $A_L(l)$ für ganzzahliges l, $l \geq 0$ das Minimum annimmt. Dann gilt auf Grund der Unimodalität der Funktion A_L $|\widehat{l}_{min} - l_{min}| < 1$. Unimodalität bei einer Funktion bedeutet, daß es mindestens ein Minimum gibt und links, bzw. rechts vom Minimum die Funktion monoton fallend, bzw. steigend ist. Weiterhin gelten auf Grund der Unimodalität für $l_{min} \geq 1$ folgende Ungleichungen:

$$\begin{aligned} A_L(l_{min}) &\leq A_L(\lceil l_{min} \rceil) &\leq A_L(l_{min} + 1) \\ A_L(l_{min}) &\leq A_L(\lfloor l_{min} \rfloor) &\leq A_L(l_{min} - 1) \end{aligned}$$

Durch Einsetzen der Werte $l_{min} + 1$ und $l_{min} - 1$ in die Funktion A_L ergibt sich:

$$A_L(l_{min} + 1) \leq 2 * A_L(l_{min}) \text{ und } A_L(l_{min} - 1) \leq 2 * A_L(l_{min})$$

Somit erhalten wir $A_L(\widehat{l}_{min}) \leq 2 * A_L(l_{min})$. Dies bedeutet, daß der Wert des ganzzahligen Minimums nahe beim Wert des reellen Minimums liegt.

Im folgenden betrachten wir nun das reellwertige Minimum ohne dabei die Abweichung zum ganzzahligen Minimum zu berücksichtigen. Die durchschnittliche Anzahl von Diskzugriffen für eine Rechteck-Schnitt-Anfrage ist somit durch $A_{L,min} * cl$ gegeben, wobei cl, $cl \geq 1$, die durchschnittliche Kettenlänge bezeichnet. Um unser Resultat in Abhängigkeit der Anzahl der Datensätze zu präsentieren, berechnen wir das übliche Leistungsmaß pm für komplexe Anfragen, wobei pm der Quotient aus der durchschnittlichen Anzahl von möglichen Kandidaten zu der durchschnittlichen Anzahl von Antworten ist. Somit erhalten wir für $2^L * emax \geq 1$ und zu einer vorgegebenen Span $(= n/(2^L * b))$

$$\begin{aligned}
pm &= \frac{(A_{L,min} * cl) * (b * Span)}{(emax + 2\sigma) * n} && (15)\\
&= \frac{(emax + 2\sigma) * 2^L * b * Span * cl + 2(emax * 2^L)^{1/2} * cl * b * Span}{(emax + 2\sigma) * n} \\
&= cl + 2(\frac{emax}{Span * b})^{1/2} * \frac{cl}{emax * 2\sigma} \frac{1}{\sqrt{n}} \\
pm &= cl + O(\frac{1}{\sqrt{n}}) && (16)
\end{aligned}$$

Dieses Resultat ist scheinbar unabhängig davon, wie hoch der Überlappungsgrad der Intervalle ist. Wir möchten hierbei aber erwähnen, daß unter den oben gemachten Voraussetzungen der durchschnittliche Überlappungsgrad der Datei $\mathcal{R}$ durch $\Omega_{\mathcal{R}} = emax * n$ gegeben ist. Die Gleichung 16 ist im wesentlichen durch die durchschnittliche Kettenlänge beeinflußt, die für Gleichverteilung sehr nahe bei 1 liegt. Nur für kleine Anfragen wirkt sich der zweite Term der Gleichung nachteilig auf die Leistung einer Anfrage aus. Der zweite Term drückt den zusätzlichen Aufwand für eine Anfrage aus, der durch den Rand des Suchbereiches verursacht wird. Man beachte dabei, daß die Ketten am Rand Intervalle enthalten, die nicht die Anfrage erfüllen.

Um einen Vergleich zu bekommen, was wir an Leistung für eine Anfrage bei einer optimalen Partitionierung gegenüber einer symmetrischen Partitionierung gewinnen können, betrachten wir die Funktion $A_L(l)$ explizit. Wir nehmen hierfür an, daß unsere Datei aus 2^L Ketten besteht, wobei eine Kette genau einem Datenblock entspricht, der vollständig mit Datensätzen gefüllt ist. Somit ergibt sich die Anzahl der Datensätze durch $n = 2^L * b$. Weiterhin beschränken wir uns in unserem Vergleich auf Punkt-Anfragen. Da eine Punkt-Anfrage ein Spezialfall der Intervall-Schnitt-Anfrage ist, können wir die oben hergeleiteten Gleichungen für $\sigma = 0$ benutzen. In Abhängigkeit von der Anzahl der Datensätze betrachten wir folgende drei Größen: $\mathbf{A_{L,min}}$, $\mathbf{A_{L,sym}} := A_L(L/2)$, und den relativen Leistungsgewinn $\mathbf{Var(L)} = (A_{L,sym} - A_{L,min}) /\ A_{L,sym}$.

In unserem ersten Diagramm in Abb. 35 haben wir die Werte $A_{L,min}$, $A_{L,sym}$ und Var(L) in Abhängigkeit der Anzahl der Datensätze veranschaulicht, wobei wir $emax = 1/256$ und $b = 50$ gewählt haben. Man beachte, daß $n = 2^L * b$ gilt. In der Abb. 35 ist der

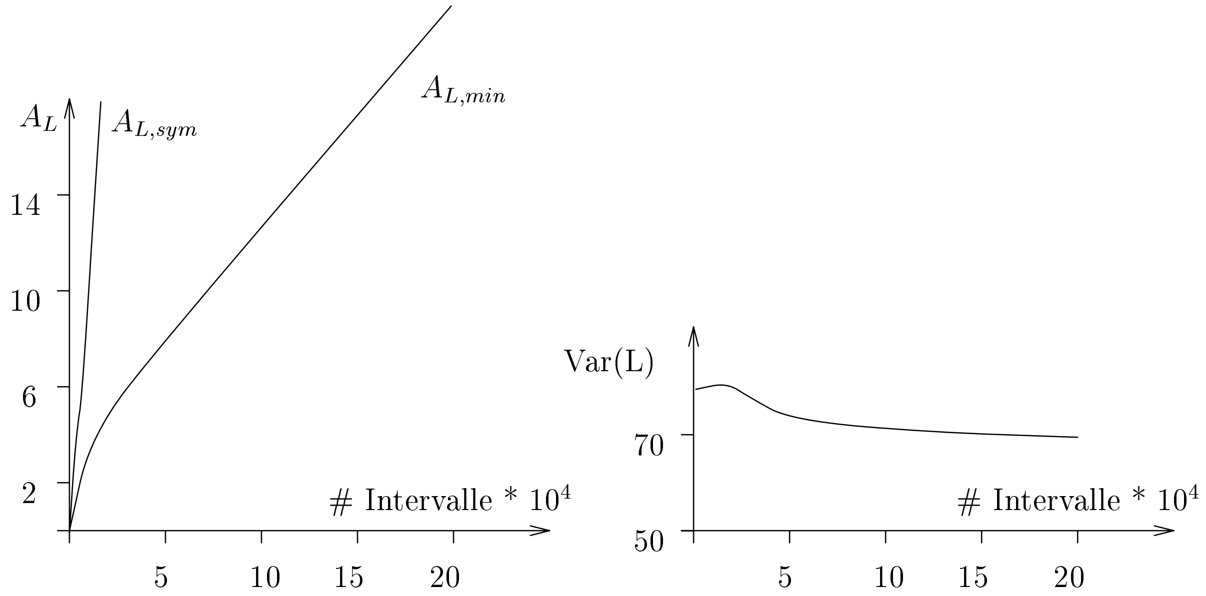


Abbildung 35: Vergleich der Leistung bei einer symmetrischen im Vergleich zu einer asymmetrischen Partitionierung

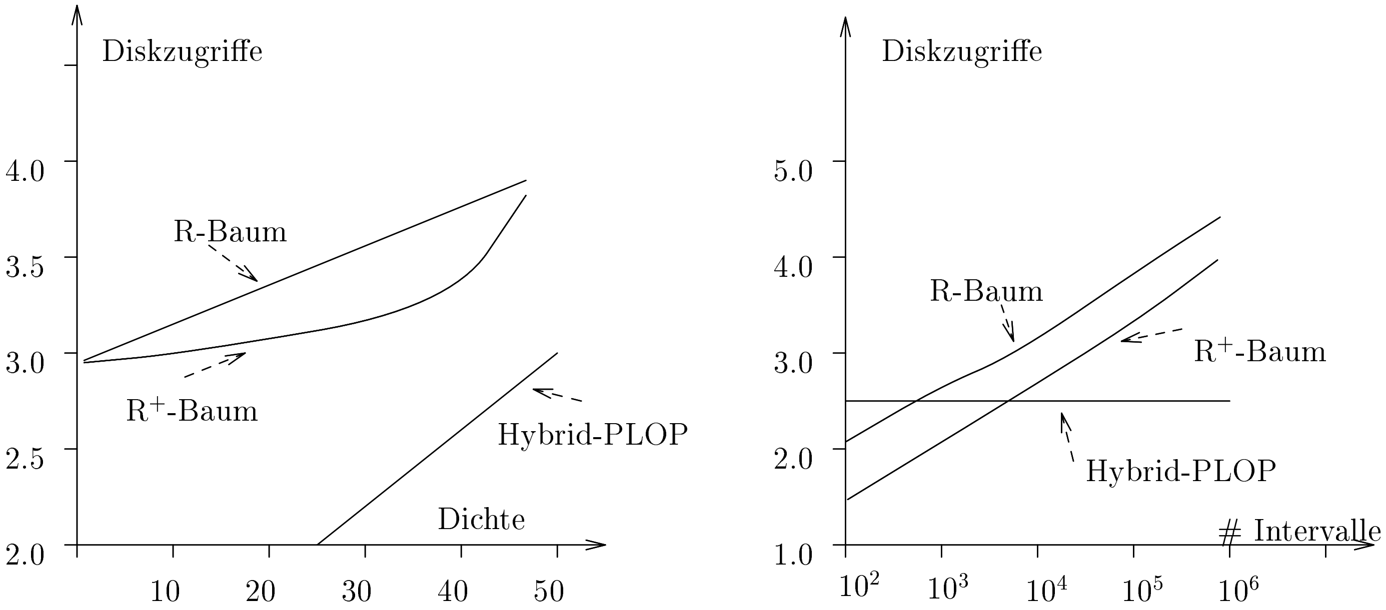


Abbildung 36: Vergleich des R-Baums,R^+-Baums und der asymmetrischen Variante von PLOP-Hashing

enorme Unterschied bei der Leistung der Verfahren aufgezeigt. So schneidet der Suchbereich z. B. nach dem Einfügen von 51200 Intervalle im Fall der optimalen Partitionierung 8 Gitterzellen und im Fall einer symmetrischen Partitionierung 36 Gitterzellen. Der relative Leistungsgewinn liegt unabhängig von der Anzahl der Intervalle bei etwa 70%.

Um unsere Variante von PLOP-Hashing mit bekannten RZS zu vergleichen, berechnen wir für die Verteilung der Intervalle, wie sie für den Leistungsvergleich von R-Baum und R^+-Baum in [FSR 87] benutzt wurde, die durchschnittliche Leistung unseres Verfahrens. Deshalb geben wir zunächst an, wie die Verteilung der Daten berechnet werden kann:

- alle Intervalle I = (c,σ) in der Datei $\mathcal{R}$ besitzen die gleiche Ausdehnung σ, $\sigma > 0$
- sei n, $n > 0$, die Anzahl der Intervalle, so werden die Zentren $c_j \in (-\sigma, 1+\sigma)$ der Intervalle $I_j = (c_j, \sigma)$ wie folgt gewählt:

$$c_j = -\sigma + (1+\sigma) * \frac{j}{n+1}\ , \quad 1 \leq j \leq n$$

Zu beachten ist, daß in [FSR 87] im Gegensatz zu unseren Voraussetzungen für diese Verteilung nicht das Einheitsintervall als Datenraum vorausgesetzt wurde.

Unter der Annahme, daß PLOP-Hashing den Datenraum beim Einfügen dieser Daten gleichmäßig unterteilt und daß die Ketten genau aus einer vollständig gefüllten Seite bestehen, haben wir die durchschnittlichen Kosten für eine Punkt-Anfrage berechnet. Wir möchten dazu bemerken, daß in [FSR 87] ebenfalls für R- und R^+-Baum eine 100% Auslastung der Seiten angenommen wurde. Für diese Verteilung ist der durchschnittliche Überlappungsgrad wie folgt gegeben:

$$\Omega = \frac{2\sigma}{(1+2\sigma)}(n+1) \tag{17}$$

In Abb. 36 auf der linken Seite haben wir für eine feste Anzahl von Datensätzen (n= 100000), die Anzahl der Diskzugriffe für eine Punkt-Anfrage in Abhängigkeit des durchschnittlichen Überlappungsgrads Ω (siehe Gleichung 17) aufgezeigt. Zu beachten ist, daß mit wachsendem Überlappungsgrad Ω nach Gleichung 17 die Ausbreitung σ der Intervalle größer werden muß. Auf der rechten Seite der Abb. 36, variiert dagegen die Anzahl der Datensätze und der Überlappungsgrad Ω, $\Omega = 40$, ist konstant. Somit muß mit wachsender Anzahl von Intervallen die Ausbreitung der Intervalle kleiner werden. In beiden Abbildungen gilt für PLOP-Hashing $emax * 2^L < 1$ und für die Situation, die zu der rechten Seite von Abb. 36 gehört, ergibt sich sogar, daß $2^L * emax \approx \Omega/b = 0.8$ nahezu konstant ist. Somit würde PLOP-Hashing in solch einer Situation nur bzgl. der Zentrumsachsen partitionieren, d. h. wir würden PLOP-Hashing nur mit der Technik der ÜR verwenden. Das wichtigste Ergebnis bei diesen Vergleichen ist, daß für große Datenbestände unser Verfahren sowohl dem R-Baum als auch dem R^+-Baum überlegen ist.

Die Ergebnisse aus diesem Abschnitt sind unter der unrealistischen Annahme der Gleichverteilung der Rechtecke gewonnen worden. Unter diesen Voraussetzungen haben wir den Vorteil einer asymmetrischen gegenüber einer symmetrischen Partitionierung analytisch aufgezeigt.

Die Gleichung 14 verdeutlicht, daß die Leistung einer Punkt-Anfrage ($\sigma = 0$) linear von emax abhängt. Nehmen wir an, daß eine Menge von Intervallen $\{I_1, \ldots, I_n\}$ verwaltet wird mit $I_j = (c_j, \sigma_j)$, $c_j \in [0, 1), \sigma_j \in [0, 0.5)$, $1 \leq j \leq n$. Weiterhin existiere genau ein $j_0 \in \{1, \ldots, n\}$ mit $\sigma_{j_0} = emax$ und $\sigma_j < emax/2$ für alle $j \in \{1, \ldots, n\} \setminus \{j_0\}$. Somit gibt es genau ein Intervall, das doppelt so lang wie alle anderen Intervalle ist. Würden wir dieses eine Intervall in zwei Intervalle halbieren, so würden wir die Leistung einer Punkt-Anfrage (nach Gleichung 14 für $\sigma = 0$) absolut um fast 100% verbessern. Dieses einfache Beispiel zeigt auf, daß ein Zerschneiden von Intervallen die Effizienz von Anfragen erheblich verbessern kann. Wie solch ein Zerschneiden aussehen kann, ohne die Nachteile der Clipping-Verfahren in Kauf zu nehmen, werden wir in Abschnitt 5.8 aufzeigen.

5.7 Wahl der Partitionierung bei Nichtgleichverteilung

Im letzten Abschnitt haben wir unter der Voraussetzung der Gleichverteilung analytisch aufgezeigt, daß eine asymmetrische Partitionierung die Leistung von PLOP-Hashing in Kombination mit der Transformationstechnik erheblich verbessern kann. Wir haben dabei die Gleichung 13 hergeleitet, die uns nach der Verdopplung der Datei die Bestimmung der neuen Splitachse ermöglicht. Daraus ergibt sich zunächst folgende Strategie:

- Sei L das Level der Datei und $l_1 = \lfloor \log_2 m_1 \rfloor$. Falls nach der Verdopplung einer Datei $A_{L+1}(l_1) < A_{L+1}(l_1 + 1)$ gilt, so ist die Ausdehnungsachse die nächste Splitachse, andernfalls ist die Zentrumsachse die nächste Splitachse.

Diese einfache Regel für die asymmetrische Partitionierung führt i. a. auch bei realistischen Datenverteilungen zu einer verbesserten Leistung der Raum-Anfragen. Da die Gleichung 13 (siehe S. 98) und somit auch diese Strategie unter der Annahme gleichverteilter Daten hergeleitet wurden, ist bei extremen Nicht-Gleichverteilungen nicht mit einer Leistungsverbesserung zu rechnen. Deshalb stellen wir einen weiteren Algorithmus zur Bestimmung der nächsten Splitachse vor, der explizit die Eigenschaften des PLOP-Hashings ausnutzt. Ziel ist es dabei, die Anzahl von Zugriffe für eine Punkt-Anfrage durch eine geeignete Wahl der Splitachse zu minimieren. Eine Verallgemeinerung auf Intervall-Schnitt-Anfragen ist in einfacher Weise zu realisieren.

Im folgenden nehmen wir an, daß unsere Datei aus 2^L Ketten besteht. Die Idee unseres Algorithmus beruht auf der Eigenschaft PLOP-Hashings, daß die Beschreibung der Partitionierung des Datenraums und somit die Beschreibung aller Seitenregionen im Hauptspeicher liegen. Wir verwenden hierbei wieder den Begriff des orthogonalen Gitters G_D wie er in Abschnitt 2.1 definiert wurde. Ausgehend von einer Zerlegung des Datenraums D in Zellen mit Hilfe eines orthogonalen Gitters $G_D(P_1, P_2)$ berechnen wir uns zu zwei geeigneten Mengen von Partitionierungspunkten Q_1 und Q_2 je eine virtuelle orthogonale Gitterpartitionierung $G_D^2(P_1, Q_2)$, bzw. $G_D^1(Q_1, P_2)$. Die Gitter G_D^2 und G_D^1 unterteilen dabei den Datenraum D in 2^{L+1} Zellen. Gegenüber dem Gitter G_D unterscheiden sie sich nur dadurch, daß die Anzahl der Scheiben (bzw. Partitionierungspunkte) in der ersten Achse für G_D^1 und in der zweiten Achse für G_D^2 verdoppelt wurden. Die Partitionierungspunkte der anderen Achsen werden dabei direkt von dem Gitter $G_D(P_1, P_2)$ übernommen. In einem weiteren Schritt berechnen wir uns dann für einige Anfragen die Anzahl der Zellen der Gitter $G_D^2(P_1, Q_2)$ und $G_D^1(Q_1, P_2)$, die den Suchbereich der Anfrage schneiden. Das Gitter mit minimaler Anzahl von Zellen in den einzelnen Suchbereichen bestimmt unsere nächste Expansionsachse. Mehr formaler können wir den Algorithmus wie folgt beschreiben:

1. Berechne neue orthogonale Gitter $G_D^2(P_1, Q_2)$ und $G_D^1(Q_1, P_2)$, wobei die Menge der Partitionierungspunkte Q_i sich mittels Interpolation aus P_i ergibt und $\mid Q_i \mid = 2 \mid P_i \mid -1$ gilt, $1 \leq i \leq 2$.

2. Für $k > 1$ bestimme Punkte $S_1, \ldots, S_k$. Berechne die Anzahl der Gitterzellen A_i^2 der Gitter $G_D^2(P_1, Q_2)$ und A_i^1 der Gitter $G_D^1(Q_1, P_2)$, die durch den Suchbereich der Punkt-Anfrage für S_i, $1 \leq i \leq k$, geschnitten werden.

3. Falls
$$\sum_{i=1}^{k}(A_i^1 - A_i^2) > 0$$
so ist die 2. Achse andernfalls die 1. Achse die nächste Splitachse.

Bemerkenswert an diesem Algorithmus ist, daß kein einziger Diskzugriff benötigt wird, da nur auf Information der binären Bäume zurückgegriffen wird, welche resident im Hauptspeicher liegen. Der Wert des Parameters k sollte dabei abhängig von der Anzahl der eingefügten Intervalle sein. Welche Werte von k genau in Frage kommen, kann nur durch praktische Tests aufgezeigt werden. Die Wahl der Punkte $S_1, \ldots, S_k$ sollte möglichst von der Verteilung der real auftretenden Anfragen abhängig sein. Ist diese Verteilung nicht bekannt, so setzen wir voraus, daß jeder Datensatz mit gleicher Wahrscheinlichkeit gesucht wird. Deshalb sollten die Punkte $S_1, \ldots, S_k$ wie die Zentren der Intervalle verteilt sein.

5.8 Kontrolliertes Clipping

Ein Nachteil bisheriger auf Clipping basierender RZS ist, daß diese speziell zur Unterstützung von Punkt- und Rechteck-Umgebungs-Anfragen konzipiert sind. Solche Anfragen können z. B. für PLOP-Hashing (in Kombination mit Clipping) durch genau einen Zugriff auf eine Kette beantwortet werden. Dagegen werden Rechteck-Schnitt-Anfragen um so ineffizienter beantwortet, desto größer das zugehörige Anfrage-Rechteck ist. Dies liegt darin begründet, daß bei sehr großen Anfrage-Rechtecken die Wahrscheinlichkeit des Zugriffs auf geclippte Rechtecke sehr hoch ist, die demselben Raumobjekt, bzw. dem MUR des Raumobjekts zugeordnet sind.

Weiterhin haben wir aufgezeigt, daß für jede auf Clipping basierende RZS, insbesondere dann, wenn der Überlappungsgrad der Raumobjekte sehr hoch ist, eine Überlauforganisation erforderlich ist.

Die Technik des *kontrollierten Clippings* verallgemeinert den Ansatz bisheriger C-Verfahren, wobei primär Rechteck-Schnitt-Anfragen unterstützt werden sollen. Hierfür setzen wir voraus, daß die durchschnittliche Ausdehnung der Anfrage-Rechtecke bekannt ist.

Im folgenden beschränken wir uns der Einfachheit halber auf Intervalle. Dabei sei σ_{avg} die durchschnittliche Ausdehnung der Anfrageintervalle. Unter diesen Voraussetzungen wäre es nicht sinnvoll, ein Intervall mit Ausdehnung kleiner σ_{avg} zu clippen, da bei einer Rechteck-Schnitt-Anfrage i. a. sonst mehrmals auf die geclippten Intervalle zugegriffen werden muß. Da aber solche Intervalle trotzdem Seitengrenzen schneiden können, müssen wir zusätzlich zu der Technik des kontrollierten Clipping noch eine andere Technik, wie z. B. Transformation oder überlappende Regionen, verwenden.

Nehmen wir an, daß wir ein Intervall $I = (c, e)$ in eine Datei einfügen sollen, wobei $e > \sigma_{avg}$ gilt. Zunächst zerteilen wir dieses Intervall I in 2^t Intervalle $S_1, \ldots, S_{2^t}$ mit

$$S_i = (c - (\frac{2i-1}{2^t} - 1) * e, \frac{e}{2^t}) \quad , 1 \leq i \leq 2^t$$

wobei t, $t \geq 1$, so bestimmt ist, daß $e * 2^{-t} < \sigma_{avg}$ und $e * 2^{-t+1} \geq \sigma_{avg}$ erfüllt ist. Sodann bestimmen wir zu den Intervallen $S_1, \ldots, S_{2^t}$ die Menge der zugehörigen Seitenadressen $\{adr_1, \ldots, adr_k\}$, $k \leq 2^t$, und fügen das ursprüngliche Intervall in die Seite adr_i, $1 \leq i \leq k$, ein. Zu beachten ist hierbei, daß geclippte Intervalle, die zu einem Raumobjekt gehören, nicht gemeinsam in einer Seite liegen, sondern daß in einer Datenseite nur genau eine Referenz zu einem Raumobjekt abgespeichert wird.

In vielen Anwendungen ist die durchschnittliche Ausdehnung eines Anfrage-Rechtecks für eine Rechteck-Schnitt-Anfrage nicht exakt bekannt. Ein ungenauer Schätzwert für σ_{avg} wirkt sich negativ auf die Leistung der RZS aus, die kontrollierten Clipping verwenden. Bisherige Clipping-Verfahren setzen im Prinzip $\sigma_{avg} = 0$ voraus. Dagegen erwarten wir in typischen Anwendungen, daß nicht Punkt-Anfragen, sondern Rechteck-Schnitt-Anfragen möglichst effizient beantwortet werden sollen.

Trotz der Abhängigkeit der Leistung von den jeweiligen Anfragen können wir durch geeignete Wahl des Clippingfaktors die Leistung unserer Hybridstruktur global erhöhen. Unter dem Clippingfaktor verstehen wir hierbei die Schranke wo Intervalle, deren Ausdehnung diesen Faktor überschreiten, geclippt werden. Wie bereits in den letzten Abschnitten aufgezeigt, hängt die Leistung unserer Hybridstruktur linear von dem Wert emax ab. Deshalb ist es sinnvoll, in Situationen wo nur wenig große aber viele kleine Intervalle in der Datei liegen, die großen Intervalle zu clippen. Dadurch wird der Parameter emax reduziert und i. a. die Anfrageleistung erhöht.

Die Entscheidung darüber, was große und was kleine Intervalle sind, kann im Prinzip nur getroffen werden, nachdem alle Intervalle eingefügt worden sind und sich die Datei in einem statischen Zustand befindet. Ähnlich zum R-Baum [RL 85] und R^+-Baum [SRF 87] können wir eine globale Reorganisation für die Datei durchführen, wo für einen realistischen Clippingfaktor die Datei durch Hybrid-PLOP neu aufgebaut wird.

Gegenüber der Technik der Transformation werden wir bei Punkt-Anfragen und Rechteck-Schnitt-Anfragen an Effizienz gewinnen. Dagegen erwarten wir für Rechteck-Inhalts-Anfragen i. a. Effizienzverluste, da die Größe des zu durchsuchenden Anfragebereichs sich durch (kontrolliertes) Clipping erhöht.

5.9 Vergleich von Raumzugriffsstrukturen

Nachdem wir in den letzten Abschnitten RZS für die Organisation von d-dimensionalen Rechtecken vorgestellt haben, wollen wir nun in einen experimentellen Vergleich die Leistung verschiedener RZS aufzeigen. Die Leistung einer RZS ist unter realistischen Voraussetzungen, ebenso wie die einer mehrdimensionalen PZS, analytisch schwierig aufzuzeigen.

Unter der Vielzahl von möglichen RZS für einen Leistungsvergleich haben wir uns auf die folgenden sechs Verfahren (V1) - (V6) beschränkt:

(V1) Clipping & PLOP-Hashing

(V2) Überlappende Regionen & PLOP-Hashing

(V3) symmetrische Transformation & PLOP-Hashing

(V4) asymmetrische Transformation & PLOP-Hashing

(V5) Mehr-Schichten-PLOP-Hashing

(V6) Transformation & Buddy-Hashbaum

Zu bemerken ist, daß für alle auf PLOP-Hashing basierenden RZS die gleichen Kontrollfunktionen **(Ex 1.0)** und **(Co 0.45)** benutzt wurden (siehe Abschnitt 2.3.1 und 2.3.2). Die T-Verfahren (V3) und (V4) basieren auf der Zentrums-Darstellung, während das Verfahren (V6) auf der Ecken-Repräsentation beruht. Für den Buddy-Hashbaum haben wir die Ecken- der Zentrumsdarstellung vorgezogen, da in allen unseren Experimenten für die Eckendarstellung weniger Zugriffe und wesentlich weniger CPU-Zeit benötigt wurde. Zu beachten ist, daß das Verfahren (V4) nicht dem in den letzten Abschnitten vorgestellten Hybrid-PLOP-Hashing entspricht. Bisher haben wir die Technik der überlappenden Regionen bei dem Verfahren (V4) nicht miteinbezogen, was in einem weiteren Vergleich noch zu realisieren wäre. Die Leistung unserer Hybridstruktur sollte aber bei optimaler Wahl der Splitachsen mindestens so effizient wie die Verfahren (V2) und (V4) sein.

Da wir bisher noch nicht näher auf die Mehr-Schichten Organisation bei RZS eingegangen sind, wollen wir hier kurz die prinzipielle Idee vorstellen. Die Motivation für solch ein Verfahren ist, die mehrfache Abspeicherung von Rechtecken, wie es bei Clipping-Verfahren vorkommt, möglichst zu reduzieren. Wir haben die Mehr-Schichten Technik auf PLOP-Hashing angewendet. Dabei wird eine Menge d-dimensionaler Rechtecke durch mehrere voneinander abhängiger d-dimenionaler PLOP-Hashing-Dateien organisiert. Jeweils eine PLOP-Hashing-Datei ist einer Schicht zugeordnet. Bei der letzten Schicht wird die Technik des Clipping auf PLOP-Hashing angewendet, während auf den anderen Schichten das gewöhnliche PLOP-Hashing die Dateien organisiert. Beim Einfügen eines Rechtecks wird, ausgehend von der ersten Schicht, eine Seite in der zugehörigen Datei gesucht, wobei das Rechteck ganz in der Seitenregion enthalten ist. Falls solch eine Seite existiert, wird das Rechteck in diese Seite eingefügt. Andernfalls ist man in der letzten Schicht angelangt, in deren zugehörige Datei das Rechteck mit Hilfe von Clipping eingefügt wird. Für eine genauere Vorstellung des Verfahren wollen wir auf [SW 88] verweisen, wo ein Mehr-Schichten Verfahren basierend auf der Gitterdatei ausführlich vorgestellt wurde. Entsprechend dem Vorschlag in [SW 88] haben wir die Organisation auf vier Schichten begrenzt.

Neben der Auswahl der RZS ist eine geeignete Auswahl von Daten für unsere Experimente besonders wichtig. Da wir trotz intensiver Bemühungen nur wenige Daten aus realen Anwendungen vorweisen können, haben wir Rechtecke synthetisch erzeugt. Mittels eines Zufallszahlengenerator haben wir uns Rechtecke erzeugt, die nicht notwendigerweise im Datenraum lagen. Alle Rechtecke, die den Rand des Datenraums schneiden wurden dabei nicht berücksichtigt. Dabei betrachten wir Dateien (A1) - (A4) mit jeweils 10,000 2-dimensionalen Rechtecken, wobei die Rechtecke wie folgt verteilt sind:

(A1) Das Zentrum ist gleichverteilt in $(0,1)^2$. Die Ausdehnung ist gleichverteilt in (0,emax) mit emax = 0.01.

(A2) Entspricht der Datei (A1) mit der Ausnahme, daß emax = 0.05 gewählt wurde.

(A3) Das Zentrum folgt einer Normalverteilung mit Erwartungswert 0.5 und Varianz 0.2. Die Ausdehnung des Rechtecks ist exponentiell verteilt in (0,emax), wobei der Erwartungswert durch $\frac{emax}{2\ln 10}$ gegeben ist und der Parameter emax = 0.02 erfüllt.

(A4) Das Zentrum eines Rechtecks ist gleichverteilt in $(0,1)^2$. Die Ausdehnung des Rechtecks folgt einer logarithmischen Verteilung mit $emax * \frac{\ln Y}{4{*}\ln 10}$, wobei $Y \in (.0001, 1.0)$ eine gleichverteilte Zufallsvariable ist und der Parameter $emax = 0.05$ gewählt wurde.

Zusätzlich zu diesen Dateien haben wir eine fünfte Datei **(A5)** mit realen Daten betrachtet, die aus den Höhenliniendaten gewonnen wurden, die wir bereits in Abschnitt 4 betrachtet haben (siehe Abb. 24). Dabei haben wir um jede Höhenlinie ein minimal umgebendes Rechteck gelegt und diese Rechtecke in der Datei **(A5)** abgespeichert. Insgesamt liegen in der Datei 1883 Rechtecke.

Bei den synthetisch erzeugten Dateien (A1) und (A2) gibt es nur wenig große und sehr viele kleine Rechtecke. Diese Eigenschaft der Daten wird z. B. in [RL 85] als realistisch angesehen. Unabhängig davon haben wir die maximale Ausdehnung eines Rechtecks bei allen Dateien relativ zu dem gesamten Datenraum niedrig gehalten. Der durchschnittliche Überlappungsgrad der Dateien ergibt sich aus der durschnittlichen Anzahl von Antworten für eine Punktanfrage. In Tab. 7 haben wir diesen Wert mittels Punktanfragen angenähert. Ein Überlappungsgrad kleiner 10 kann in Anwendungen, wie z. B. bei der Abspeicherung von Parzellengrenzen umgebender Rechtecken, als realistisch angesehen werden.

Für die Dateien (A1) - (A5) haben wir jeweils $\lceil \frac{n}{500} \rceil$ Punkt-Anfragen (**PA**) und Rechteck-Schnitt-Anfragen (**RSA**) beantworten lassen. Dabei haben wir $\lceil \frac{n}{500} \rceil$ Rechtecke aus der jeweiligen Datei ausgewählt. Durch die Zentren der Rechtecke haben wir die Anfragepunkte einer PA bestimmt, während bei einer RSA die Rechtecke selbst als Anfrage-Rechtecke dienten. Die durchschnittliche Anzahl von Antworten pro Anfrage haben wir in der Tabelle 7 aufgezeigt. Zu bemerken ist, daß unsere Anfragerechtecke für die RSA relativ klein gehalten sind. Ähnlich zu PZS hängt die Leistung von RSA bei großen Anfrage-Rechtecken im wesentlichen von der Span der Verfahren ab. Deshalb ist eine **hohe Span** bei RZS von zentraler Bedeutung.

Bevor wir nun auf die Ergebnisse unseres Vergleichs eingehen, wollen wir ein weiteres interessantes Resultat vorstellen. Solange über die Transformation von 2-dimensionalen

Datei	PA	RSA
(A1)	1.0	1.0
(A2)	5.0	12.5
(A3)	2.3	9.6
(A4)	1.9	5.3
(A5)	11.0	27.8

Tabelle 7: Anzahl der Antworten bei Punkt- und Rechteck-Schnitt-Anfragen für die jeweiligen Dateien (A1) - (A5)

Rechtecke in 4-dimensionale Punkte diskutiert wird, ist die Frage von Interesse, welche Repräsentation der Punkte - unabhängig von der darunterliegenden PZS - am effizientesten ist. Da bei früheren PZS wie z. B. bei der Gitterdatei [NHS 84], die Verteilung der Daten wesentlich die Leistung einer PZS beeinflußte, erschien die Zentrums-Repräsentation von Rechtecken bisher am geeignetesten. Die Leistung des (gepackten) Buddy-Hashbaums (BHB) wird kaum durch die Verteilung der Daten beeinflußt. Daher erscheint uns der BHB eine geeignete mehrdimensionale PZS zu sein, um diese Frage zu beantworten. In Tab. 8 haben wir nun die Anzahl der Zugriffe notiert, die der BHB bei der Ecken- und Zentrums-Repräsentation für die Beantwortung der Anfragen bei den Dateien (A1)-(A5) benötigt hat. Hierbei zeigt sich klar, daß die Ecken-Repräsentation die geeignetere Punktdarstellung für Rechtecke ist. Ein Grund hierfür ist, daß bei der Ecken-Repräsentation die Begrenzungslinien der Anfragebereiche parallel zu den Partitionierungslinien des BHB liegen und somit der Rand des Anfragebereichs i. a. weniger Seiten als bei der Zentrums-Repräsentation schneidet. Wir möchten aber hierzu anmerken, daß für eine Hybrid-Struktur nur die Zentrums-Repräsentation in Frage kommt.

In den Tabellen 9 - 13 haben wir die Ergebnisse unseres Vergleichs der RZS notiert. Neben *RSA* und *PA* haben wir in diesen Tabellen die Abkürzung IO-Aufbau für die Anzahl der Diskzugriffe zum Aufbau der Datei verwendet. In Tab. 7 haben wir die durchschnittliche Anzahl von Antworten für die Anfragen notiert. Wir können nun die Resultate wie folgt zusammenfassen:

Datei	Punkt-Anfrage		Rechteck-Schnitt-Anfrage	
	Zentrums-Rep.	Ecken-Rep.	Zentrums-Rep.	Ecken-Rep.
(A1)	757	253	789	268
(A2)	738	344	876	441
(A3)	719	343	808	358
(A4)	520	277	552	282
(A5)	54	43	59	50

Tabelle 8: Vergleich der Ecken- und Zentrums-Repräsentation für den gepackten Buddy-Hashbaum

RZS	Span	IO-Aufbau	PA	RSA
(V1)	0.37	35,525	27	40
(V2)	0.53	28,139	57	64
(V3)	0.49	48,967	730	756
(V4)	0.53	28,105	38	38
(V5)	0.43	43,786	136	151
(V6)	0.72	27,712	253	268

Tabelle 9: Vergleich der Leistung von RZS für die Datei (A1)

- Es ist ein klarer Unterschied bei der Leistung der Verfahren (V3) und (V4) zu erkennen. Bis auf das Ergebnis in Tab. 13 ist die asymmetrische der symmetrischen Partitionierung des Datenraums vorzuziehen.

- Obwohl der BHB eine sehr effiziente PZS ist, die Span des BHB sehr hoch liegt und die Aufbaukosten beim BHB am geringsten sind, ist die zugehörige RZS (V6) nicht die effizienteste bei der Beantwortung der Anfragen. Auf Grund der hohen Span erwarten wir aber bei größeren Anfrage-Rechtecken ein gegenüber den anderen Verfahren verbessertes Leistungsverhalten.

- Gemittelt über alle Experimente scheinen die Verfahren (V1) (Clipping & PLOP-Hashing), (V2) (Überlappende Regionen & PLOP-Hashing) und (V5) (Mehr-Schichten PLOP-Hashing) Anfragen am effizientesten zu beantworten. Auf Grund der z. T. niedrigen Span bei den Verfahren (V1) und (V5) erwarten wir, daß RSA mit größeren Anfrage-Rechtecken wesentlich ineffizienter beantwortet werden. Im Gegensatz dazu bleibt die Leistung des Verfahrens (V2) auch für solche RSA erhalten.

- Das Verfahren (V5) (Mehr-Schichten PLOP-Hashing), das eine Verallgemeinerung des Verfahrens (V1) (Clipping & PLOP-Hashing) darstellt, vermeidet nicht dessen wesentliche Probleme. In den Fällen (siehe Tab. 10, 13), in denen das Clipping-Verfahren (V1) eine niedrige Span aufweist, ist bei dem Mehr-Schichten-Verfahren ein entsprechendes Verhalten festzustellen. Für diese Dateien speichert das Verfahren (V5) die meisten Rechtecke in der letzten Schicht ab, deren zugehörige RZS entsprechend dem Verfahren (V1) organisiert ist.

- Bisher haben wir noch nicht die Hybridstruktur implementiert, die die Technik der (asymmetrischen) Transformation und der überlappenden Regionen gemeinsam benutzt. Stattdessen haben wir zwei Spezialfälle der Hybridstruktur, nämlich die Verfahren (V2) und (V4), in den experimentellen Vergleich aufgenommen. Somit muß die Hybridstruktur bei optimaler Wahl der Splitachsen eine Leistung besitzen, die mindestens so gut ist wie die Leistung der Verfahren (V2) oder (V4).

RZS	Span	IO-Aufbau	PA	RSA
(V1)	0.10	90,100	28	161
(V2)	0.53	28,166	134	176
(V3)	0.49	49,498	992	1158
(V4)	0.60	26,473	578	915
(V5)	0.18	278,960	113	244
(V6)	0.69	27,674	344	441

Tabelle 10: Vergleich der Leistung von RZS für die Datei (A2)

RZS	Span	IO-Aufbau	PA	RSA
(V1)	0.34	36,500	24	69
(V2)	0.60	23,778	113	146
(V3)	0.44	33,700	502	584
(V4)	0.49	32,307	226	288
(V5)	0.46	42,746	102	157
(V6)	0.72	27,910	343	358

Tabelle 11: Vergleich der Leistung von RZS für die Datei (A3)

RZS	Span	IO-Aufbau	PA	RSA
(V1)	0.41	31,529	28	42
(V2)	0.64	22,837	104	133
(V3)	0.58	27,447	436	479
(V4)	0.47	35,554	344	389
(V5)	0.56	31,277	99	120
(V6)	0.76	27,702	277	282

Tabelle 12: Vergleich der Leistung von RZS für die Datei (A4)

RZS	Span	IO-Aufbau	PA	RSA
(V1)	0.12	14,013	5	39
(V2)	0.51	5,119	22	43
(V3)	0.49	7,513	78	86
(V4)	0.51	6,137	95	113
(V5)	0.37	6,854	19	34
(V6)	0.72	3,412	43	50

Tabelle 13: Vergleich der Leistung von RZS für die Datei (A5) bei sortierter Eingabe von 1883 Datensätzen

6 Zusammenfassung und Ausblick

In dieser Arbeit haben wir uns mit mehrdimensionalen Zugriffsstrukturen beschäftigt, die ein effizientes Suchen bzgl. mehrerer Schlüssel erlauben. Dabei haben wir unterschieden zwischen mehrdimensionalen Punktzugriffsstrukturen (PZS) und mehrdimensionalen Raumzugriffsstrukturen (RZS).

In den ersten Abschnitten der Arbeit haben wir zwei mehrdimensionale PZS im Detail vorgestellt: PLOP-Hashing und den Buddy-Hashbaum. PLOP-Hashing ist ein mehrdimensionales dynamisches Hashverfahren ohne Directory, welches das Konzept von linearem Hashing für mehrdimensionale Daten verallgemeinert. Im Gegensatz zu anderen PZS ohne Directory partitioniert PLOP-Hashing den Datenraum durch ein orthogonales Gitter, das sich der Verteilung der Daten anpaßt. Auf Grund der Gitterpartitionierung kann PLOP-Hashing nur für nicht bzw. schwach korrelierte Daten genügend effizient sein. Im Gegensatz dazu verschlechtert sich die Leistung von PLOP-Hashing bei stark korrelierten Daten erheblich.

Das Ziel beim Entwurf des Buddy-Hashbaums (BHB) war, eine mehrdimensionale PZS zu entwicklen, deren Leistung unabhängig von der Verteilung und dem Korrelationsgrad der Daten ist. Dabei haben wir aufgezeigt, daß das Einfügen eines Datensatzes, das Löschen eines Datensatzes und die Suche nach genau einem Datensatz auf einen Pfad des BHB beschränkt ist. Die Höhe des BHB ist dabei durch eine Funktion beschränkt, die von der Auflösung des Datenraums abhängt und die insbesondere von der Verteilung der Datensätze unabhängig ist. In einem Vergleich mit PLOP-Hashing und der 2-Level-Gitterdatei [Hin 85] haben wir die Überlegenheit des BHB gegenüber diesen PZS aufgezeigt. In einem weitergehenden Vergleich [KSSS 89] mit dem BANG-File [Fre 87] und dem hB-Baum [LS 87] bestätigen erste Resultate die Überlegenheit des BHB sogar gegenüber diesen PZS. Das Prinzip des BHB ist es, den Datenraum nicht vollständig zu partitionieren, sondern Seitenregionen von Daten- und Directoryseiten möglichst klein zu halten. Weiterhin haben wir aufgezeigt, daß die gepackte Variante des BHB eine sehr hohe Span garantiert.

Im zweiten Teil der Arbeit haben wir mehrdimensionale RZS genauer untersucht. Zunächst haben wir eine Klassifizierung existierender RZS vorgenommen. Dabei haben wir aufgezeigt, daß RZS, die Raumobjekte durch minimal umgebende Rechtecke (*MUR*) approximieren, im wesentlichen auf einer mehrdimensionalen PZS und einer der drei Techniken Clipping, überlappende Regionen und Transformation basieren. An Hand von PLOP-Hashing haben wir die zu diesen Techniken gehörenden RZS vorgestellt.

Darüber hinaus haben wir eine Hybrid-Struktur vorgeschlagen, die diese drei Techniken gemeinsam nutzt. In Abhängigkeit der Verteilung der MUR und der Dichte der Raumobjekte im Datenraum, sollen dabei die einzelnen Techniken so gewichtet werden, daß wir eine möglichst effiziente RZS erhalten. Da das einfache Konzept des Clippings im wesentlichen nur Punkt-Anfragen effizient unterstützt, haben wir stattdessen vorgeschlagen, die Technik des *kontrollierten Clippings* zu benutzen.

Schließlich haben wir die Leistung verschiedener RZS mit Hilfe von Implementierungen experimentell miteinander verglichen. Dabei hat sich gezeigt, daß Clipping-Verfahren für Punkt-Anfragen und Rechteck-Schnitt-Anfragen mit kleinen Anfrage-Rechtecken erwartungsgemäß die beste Leistung besitzen. Der Preis für diese niedrigen Suchkosten muß aber bereits bei einem niedrigen Überlappungsgrad der Objekte durch eine sehr schlechte Span bezahlt werden. Dies führt insbesondere zu einem ineffizienten Verhalten bei Rechteck-Schnitt-Anfragen mit großen Anfrage-Rechtecken. Insgesamt hat der Leistungsvergleich aufgezeigt, daß unsere Hybrid-Struktur ein interessanter und vielversprechender Ansatz für eine effiziente RZS ist.

Unsere weiteren Ziele können wir wie folgt zusammenfassen:

- Für den BHB haben wir bisher nur eine Prototypimplementierung erstellt. In Zukunft wollen wir die Implementierung des BHB um die Konzepte vervollständigen, die wir in unserer Arbeit vorgestellt haben.

- Da der BHB eine sehr effiziente PZS darstellt, ist er geradezu als Basis einer RZS prädestiniert. Insbesondere die Anwendung des Konzepts der minimalen Regionen erscheint uns das geeignete Mittel für den Entwurf von RZS zu sein. Auch ist zu klären, inwieweit eine Hybridmethode auf der Basis des BHB effizient zu realisieren ist.

- Den Leistungsvergleich aus Abschnitt 5.9 wollen wir um weitere RZS, wie den R-Baum und den R$^+$-Baum, erweitern

- Rasterapproximationen wie sie in [OM 88] für die Approximation von Raumobjekten vorgestellt wurden, können statt in einem B^+-Baum auch in einem BHB abgelegt werden. Vorteil wäre hierbei eine kompaktere Repräsentation der Rasterapproximation.

- Weiterhin wollen wir das im Abschnitt 5.2 vorgestellte Konzept des Anfrageprozessors realisieren.

- Zudem planen wir die Integration von mehrdimensionalen Zugriffsstrukturen in neuartige Datenbanksysteme, wie z. B. DASDBS [PSSWD 87] oder POSTGRES [Sto 86]. Damit verbunden ist die Entwicklung von "concurrency control" Mechanismen für PLOP-Hashing und den BHB.

A Literaturverzeichnis

[BM 72] R. Bayer, E. M. McCreight: 'Organization and maintenance of large ordered indices', Acta Informatica 1, 3, 173 -189, 1972

[BKSS 90] N. Beckmann, H. P. Kriegel, R. Schneider, B. Seeger: 'The R*-tree: an efficient and robust access method for points and rectangles', tech. report 2/90, Univ. of Bremen, 1990

[Ben 75] J. L. Bentley: 'Multidimensional search trees used for associative searching', Communications of ACM, Vol. 18, No. 9, 509-517, 1975

[Ben 79] J. L. Bentley: 'Multidimensional binary search trees in Database Applications', IEEE Trans. on Software Engineering, Vol.5, No.4, 33-340, 1979

[Bur 83] W. A. Burkhard: 'Interpolation-based index maintenance', BIT 23, 274-294, 1983

[Bur 84] W. A. Burkhard: 'Index maintenance for nonuniform record distributions', Proc. 3rd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems,173-180, 1984

[Com 79] D. Comer: 'The Ubiquitous B-tree', Computing Surveys, Vol. 11, No. 2, 121-137, 1979

[Ell 87] C. S. Ellis: 'Concurrency in linear hashing' ACM Trans. on Database Systems, Vol. 12, 2, 195-217, 1987

[FNPS 79] R. Fagin, J. Nievergelt, N. Pippenger, R. Strong: 'Extendible hashing - a fast access method for dynamic files', ACM Trans. on Database Systems, Vol. 4, 3, 315-344, 1979

[FSR 87] C. Faloutsos, T. Sellis, N. Roussopoulos: 'Analysis of object oriented spatial access methods', Proc. ACM SIGMOD Int. Conf. on Management of Data, 426-439, 1987

[FR 87] C. Faloutsos, W. Rego: 'A Grid File for Spatial Objects', technical report, CS-TR-1829, University of Maryland, April 1987, erscheint in Information Science

[FB 74] R. A. Finkel, J. L. Bentley: 'Quad trees - a data structure for retrieval on compositive keys', Acta Informatica 4, 1-9, 1974

[Fra 87] S. Frank: 'Multidimensionale dynamische Hash-Baumverfahren', Diplomarbeit, Universität Würzburg, 1987

[Fre 87] M. Freeston: 'The BANG file: a new kind of grid file', Proc. ACM SIGMOD Int. Conf. on Management of Data, 260-269, 1987

[Fre 88] M. Freeston: 'Grid Files for Efficient Prolog Clause Access', technischer Bericht, 1988

[Gre 89] D. Greene: 'An implementation and performance analysis of spatial data access methods', Proc. 5th Int. Conf. on Data Engineering, 606-615, 1989

[Gün 88] O. Günther: 'The cell tree: an index for geometric databases', technical report TR-88-002, Int. Computer Science Institute, Berkeley, 1988

[Güt 88] R.H. Güting: 'Geo-relational algebra: a model and query language for geometric database systems', Proc. Int. Conf. on Extending Database Technology, Venice, 506-527, 1988

[Gut 84] A. Guttman: 'R-trees: a dynamic index structure for spatial searching', Proc. ACM SIGMOD Int. Conf. on Management of Data, 47-57, 1984

[HKSS 88] S. Heep, H.P. Kriegel, R. Schneider, B. Seeger: 'Konzepte zur Suche geometrisch ähnlicher Bauteile', Proc. GI Fachgespräch Non-Standard Datenbanken für Anwendungen der graphischen Datenverarbeitung Informatik-Fachberichte 171, 166-183, 1988

[Hin 85] K. Hinrichs: 'The grid file system: implementation and case studies for applications', Dissertation Nr. 7734, Eidgenössische technische Hochschule (ETH) Zürich, 1985

[HSW 88a] A. Hutflesz, H.-W. Six, P. Widmayer: 'Globally order preserving multidimensional linear hashing', Proc. Int. Conf. on Data Engineering, 1988

[HSW 88b] A. Hutflesz, H.-W. Six, P. Widmayer: 'Twin grid files: space optimizing access schemes', Proc. ACM SIGMOD Int. Conf. on Management of Data, 183-190, 1988

[KW 87] A. Kemper, M. Wallrath: 'An analysis of geometric modeling in database systems', ACM Comp. Surveys Vol. 19, No. 1, 47-91, 1987

[Kri 84] H.P. Kriegel: 'Performance comparison of index structures for multikey retrieval', Proc. ACM SIGMOD Int. Conf. on Management of Data, 186-196, 1984

[KS 86] H.P. Kriegel, B. Seeger: 'Multidimensional order preserving linear hashing with partial expansions', Proc. Int. Conf. on Database Theory, Lecture Notes in Computer Science 243, 203-220, 1986

[KS 87] H.P. Kriegel, B. Seeger: 'Multidimensional quantile hashing is very efficient for non-uniform distributions', Proc. 3rd Int. Conf. on Data Engineering, 10-17, 1987, extended version will appear in Information Science

[KS 88] H.P. Kriegel, B. Seeger: 'PLOP-Hashing: a grid file without directory', Proc. 4th Int. Conf. on Data Engineering, 369-376, 1988

[KSSS 89] H.P. Kriegel, M. Schiwietz, R. Schneider, B. Seeger, : 'Performance comparison of multidimensional point access methods', eingereicht zur Veröffentlichung, 1989

[Lar 80] P.Å. Larson: 'Linear hashing with partial expansions', Proc. 6th Int. Conf. on Very Large Databases, 224-232, 1980

[Lar 85] P.Å. Larson: 'Linear hashing with overflow handling by linear probing', ACM Trans. on Database Systems,10, 1, 75-89, 1985

[LY 81] P. Lehman, S. Yao: 'Efficient locking for concurrent operations on B-trees, ACM Trans. on Database Systems, 6,4, 650-670, 1981

[Lit 80] W. Litwin: 'Linear hashing: a new tool for file and table addressing', Proc. 6th Int. Conf. on Very Large Databases , 212-223, 1980

[LR 82] J. Lloyd, K. Ramamohanarao: 'Partial match retrieval for dynamic files', BIT, Vol. 22, 150-168, 1982

[Lom 87] D.B. Lomet: 'Partial expansion for file organization with an index', ACM Trans. on Database Systems, Vol. 12, 1, 38-71, 1987

[LS 87] D.B. Lomet, B. Salzberg: 'The hB-tree: a robust multi-attribute indexing method', technical report TR-87-05, Wang Institute of Graduate Studies, 1987

[MOD 87] F. Manola, J. Orenstein, U. Dayal: 'Geographic information processing in the PROBE database system', Proc. 8th Int. Symposium on Automation in Cartography, Baltimore, 1987

[MT 83] M. Mantyla, M. Tamminen: 'Localized set operations for solid modeling', Computer Graphics, 17, 3, 279-288, 1983

[Mei 86] A. Meier: 'Erweiterung relationaler Datenbanksysteme für technische Anwendungen, Habitilationsschrift, Eidgenössische technische Hochschule (ETH) Zürich, 1986

[NS 88] E. Neuhold, M. Stonebraker: 'Future Directions in DBMS Research', technischer Bericht TR-88-001, Int. Computer Science Institute, Berkeley, 1988

[NHS 84] J. Nievergelt, H. Hinterberger, K.C. Sevcik: 'The grid file: an adaptable, symmetric multikey file structure', ACM Trans. on Database Systems, Vol. 9, 1, 38-71, 1984

[NH 85] J. Nievergelt, K. Hinrichs: 'Storage and access structures for geometric data bases', Proc. Int. Conf. on Foundations of Data Organization, Kyoto, 335-345, 1985

[Ooi 87] B.C. Ooi: 'A data structure for geographic database', Proc. GI-Fachtagung "Datenbanksysteme in Büro, Technik und Wissenschaft", Informatik Fachbericht 136, 247-258, 1987

[OsS 83] Y. Oshawa, M. Sakauchi: 'The BD-tree - a new n-dimensional data structure with highly efficient dynamic characteristics, Proc. IFIP 9th World Computer Congress, Paris, 539-544, 1983

[Ore 82] J. A. Orenstein: 'Multidimensional tries used for associative searching', Information Processing Letters, 14, 4, 150-157, 1982

[Ore 83] J. A. Orenstein: 'A dynamic hash file for random and sequential accessing', Proc. Int. Conf. on Very Large Databases , 132-141, 1983

[OM 84] J.A. Orenstein, T.H. Merett: 'A class of data structures for associative searching', Proc 3th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 181-190, 1984

[OM 88] J. A. Orenstein, F. Manola: 'PROBE spatial data modeling and query processing in an image database application', IEEE Transaction on Software Engineering, Vol. 14, No. 5, 611-629, 1988

[Oto 84] E. J. Otoo: 'A mapping function for the directory of a multidimensional extendible hashing', Proc. 10th Int. Conf. on Very Large Databases, 491-506, 1984

[Oto 85] E. J. Otoo: 'A multidimensional digital hashing scheme for files with composite keys', Proc. ACM SIGMOD Int. Conf. on Management of Data, 214-229, 1985

[Oto 86] E. J. Otoo, : 'Balanced multidimensional extendible hash tree', Proc. 5th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 100-113, 1986

[Ouk 85] M. Ouksel: 'The interpolation based grid file', Proc. 4th ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 20-27, 1985

[OS 83] M. Ouksel, P. Scheuermann: 'Storage mappings for multidimensional linear dynamic hashing' Proc. 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems, 90-105, 1983

[PSSWD 87] H.-B. Paul, H.-J. Schek, M. H. Scholl, G. Weikum, U. Deppisch: 'Architecture and Implementation of the Darmstadt Database Kernel System', Proc. Int. Conf. ACM SIGMOD, 196-207, 1987

[Reg 85] M. Regnier: 'An analysis of the grid file algorithms', BIT 25, 335-357, 1985

[RH 86] E. M. Reingold, W. J. Hansen: 'Data Structures in Pascal', Little, Brown, 1986

[Req 80] A. A. G. Requicha: 'Representation for rigid solids: theory, methods and systems', ACM Computing Surveys, Vol. 12, 4, 437-463, 1980

[Rob 81] J. T. Robinson: 'The K-D-B-tree: a search structure for large multidimensional dynamic indexes', Proc. ACM SIGMOD Int. Conf. on Management of Data, 10-18, 1981

[RS 84] W. Romamohanarao, R. Sacks-Davis: 'Recursive Linear Hashing', ACM Trans. on Database Systems, 9, 3, 369-391, 1984

[RL 85] N. Roussopoulos, D. Leifker: 'Direct spatial search on pictorial databases using packed R-trees', Proc. ACM SIGMOD Int. Conf. on Management of Data, 17-31, 1985

[Sal 86] Salzberg, B.: 'Grid file concurrency', Information Systems, Vol. 11, 3, 235-244, 1986

[Sam 85] H. Samet: 'The quadtree and related data structures', Computing Surveys, Vol 16, No. 2, 187-260, 1984

[SO 82] P. Scheuermann, M. Ouksel: 'Multidimensional B-trees for associative searching in database systems', Information Systems, Vol. 7, No. 2, 123-137, 1982

[SK 88] B. Seeger, H. P. Kriegel: 'Design and implementation of spatial access methods', Proc. 14th Int. Conf. on Very Large Databases, 360-371, 1988

[SRF 87] T. Sellis, N. Roussopoulos, C. Faloutsos: 'The R^{+}-tree: a dynamic index for multi-dimensional objects', Proc. 13th Int. Conf. on Very Large Database , 1987

[SW 86] H.-J. Schek, W. Waterfeld 'A database kernel system for geoscientific applications', Proc. 2nd Int. Symposium on Spatial Data Handling, Seattle, 1986, 273-288

[SW 88] H.-W. Six, P. Widmayer: 'Spatial Searching in Geometric Databases', Proc. Int. Conf. on Data Engineering, 496-503, 1988

[Stö 87] H.-W. Stöppler: 'Die "Automatisierte Liegenschaftskarte" (ALK) - Überblick', Nachrichten aus dem öffentlichen Vermessungsdienst Nordrhein-Westfalen, Vol. 20, 2-3, 64-89, 1987

[Sto 86] M. Stonebraker, L. Rowe: 'The design of POSTGRES', Proc. ACM SIGMOD Int. Conf. on Management of Data, 340-355, 1986

[SRG 83] M. Stonebraker, B. Rubenstein, A. Guttman: 'Application of abstract data types and abstract indices to CAD data bases', Proc. ACM SIGMOD Conf. on Engineering Design Applications, 1983

[Tam 82] M. Tamminen: 'The extendible cell method for closest point problems, BIT 22, 27-41, 1982

[TS 82] M. Tamminen, R. Sulonen: 'The Excell method for efficient geometric access to data', Proc. 19th ACM Design Automation Conf. , 345-351, 1982

[WK 85] K.-Y. Whang, R. Krishnamurthy: 'Multilevel grid files', technischer Bericht, IBM Research Lab., Yorktown Heights, 1985

[Wir 85] Wirth N.: 'Programming in Modula-2', Springer-Verlag, 1985

[WB 87] C. T. Wu, W. A. Burkhard: 'Associative searching in multiple storage units', ACM Trans. on Database Systems, 12, 1, 38-64, 1987

B Abkürzungsverzeichnis

Abb.	Abbildung
b	Seitenkapazität einer Datenseite
bzgl.	bezüglich
BANG-File	balanced and nested grid file
BHB	Buddy-Hashbaum
BMEH	balanced multidimensional extendible hash tree
B_i	binärer Baum der i-ten Achse, $i \in \{1, \ldots, d\}$
cl	durchschnittliche Anzahl von Datenseiten pro Kette
C-Verfahren	auf der Technik des Clipping basierende RZS
d	Dimension der Datensätze
DLA	dynamische lexikographische Adreßfunktion
d. h.	daß heißt
$D_i = [min_i, max_i)$	i-ter Wertebereich oder i-te Achse des Datenraums, $i \in \{1, \ldots, d\}$
D $= (D_1, \ldots, D_d)$	Datenraum
$K = (K_1, \ldots, K_d) \in D$	d-dimensionaler Schlüssel eines Datensatzes
emax	maximale Ausdehnung eines Rechtecks im Datenraum D
ep	Expansionszeiger
$G_D = G_D(\overline{P}_1, \ldots, \overline{P}_d)$	orthogonales Gitter
hB-Baum	holey brick tree
H $=(H_0, H_1, \ldots)$	Splitgeschichte
$H_i(A)$	orthogonale Hyperebene der i-ten Achse zu dem Punkt $A \in P_i$, $i \in \{1, \ldots, d\}$
i. a.	im allgemeinen
$\vec{i}$	$\vec{i} := (i_1, \ldots, i_d)$ mit $i_j \geq 0$, $j \in \{1, \ldots, d\}$
lf(i,j)	Belegungsfaktor der Scheibe S(i,j), $1 \leq i \leq m_i$, $j \in \{1, \ldots, d\}$
L	Level der Datei
LA	lexikographische Adressfunktion
LH	lineares Hashing
m	Anzahl der Datenseiten in einer Datei
MDH	mehrdimensionale Hashverfahren
MEH	mehrdimensionales erweiterbares Hashing
MUR	minimal umgebendes Rechteck eines Raumobjekts, das achsenparallel im Datenraum D liegt
m_i	Anzahl der Partitionierungspunkte der i-ten Achse ($m_i =\mid P_i \mid$), $i \in \{1, \ldots, d\}$
n	Anzahl der Datensätze bzw. Datenobjekte in einer Datei
Ω	Dichte einer Menge $\mathcal{R}$ von Rechtecken, $\Omega := \Omega_{\mathcal{R}}$

OId_i	Objektidentifikator des Objekts O_i, $i \in \{1, \ldots, n\}$
O_i	Raumobjekt, $i \in \{1, \ldots, n\}$
PA	Punkt-Anfrage
PLOP-Hashing	piecewise linear order preserving hashing
P_i	Menge der zur i-ten Achse gehörenden Partitionierungspunkte, $i \in \{1, \ldots, d\}$
$\overline{P}_i$	$= P_i \cup \{min_i, max_i\}$
PZS	Punktzugriffsstruktur
RSA	Rechteck-Schnitt-Anfrage
RZS	Raumzugriffsstruktur
s	Splitachse, bzw. Mischachse $s \in \{1, \ldots, d\}$
Span	Speicherplatzausnutzung
S(i,j)	Scheibe zu dem Index i, $1 \leq i \leq m_i$, in der j-ten Achse, $j \in \{1, \ldots, d\}$
T-Verfahren	auf der Technik der Transformation basierende RZS
ÜR-Verfahren	auf der Technik der überlappenden Regionen basierende RZS
w	Mächtigkeit des Datenraums D, $2^w = \mid D \mid$
2LGF	2-Level Gitterdatei (2-level grid file)
z	Mächtigkeit der i-ten Achse, $2^z = \mid D_i \mid$, $i \in \{1, \ldots, d\}$
z. B.	zum Beispiel
z. T.	zum Teil

Tabellarischer Lebenslauf

9. Mai 1960	geboren in Nieder-Klingen (Kreis Darmstadt-Dieburg)
Aug. 66 - Juni 70	Besuch der Grundschule in Nieder-Klingen
Sep. 70 - Juli 72	Besuch der Förderstufe an der Otzbergschule in Lengfeld
Sep. 72 - Juni 79	Besuch des Max-Planck-Gymnasiums in Groß-Umstadt
Juni 79	Abitur
Jul. 79 - Sep. 80	Grundwehrdienst
Okt. 80 - März 83	Studium der Mathematik an der TH Darmstadt
April 83 - Feb. 86	Fortführung des Studiums an der Universität Würzburg
Feb. 86	Diplom der Mathematik
April 86 - Sep. 86	Wissenschaftlicher Mitarbeiter am Institut für Informatik II der Universität Karlsruhe
Okt. 86 - März 87	Wissenschaftlicher Mitarbeiter an der Universität Würzburg in dem DFG Projekt ”Indexstrukturen für Geo-Daten”
April 87 - Aug. 89	Wissenschaftlicher Mitarbeiter an der Universität Bremen in dem DFG Projekt ”Indexstrukturen für Geo-Daten”
ab Aug. 89	Wissenschaftlicher Assistent im Studiengang Informatik an der Universität Bremen