Ausarbeitung zum Vortrag
"2d Bereichssuche im externen Speicher: Untere Schranken und effiziente Spezialfälle"

Eric Schwarzkopf
eschwarzkopf@acm.org

1 Einleitung

Die effiziente Verwaltung und der schnelle Zugriff auf multidimensionale Daten (Daten mit mehr als einem Schlüssel) ist für viele Anwendungen, von GIS bis zu System zur visuellen Wahrnehmung von Objekten, äußerst wichtig. Eine spezielle Aufgabe, die multidimensionale Datenbanken erfüllen müssen, ist die Bereichssuche über einer Anzahl von Schlüssel. Entsprechend wurden eine Reihe von Datenstrukturen und Algorithmen entwickelt, die diese Art von Suche effizient unterstützen (einen Überblick bietet z.B. [5]).

Wir werden uns hier mit dem Problem zweidimensionalen Bereichsanfragen auseinandersetzen. Etwas genauer: Wir betrachten eine Punktemenge P mit $P \subset \mathbb{R}^2$ und wollen Anfragen der Form *"Welche Punkte der Form (x, y) in P erfüllen für a, b, c, $d \in \mathbb{R}$ die Bedingungen $a \leq x \leq b$ und $c \leq y \leq d$?"* effizient beantworten.

Man würde sich wünschen, diese Anfragen ähnlich platz- und zeiteffizient wie eindimensionale Bereichssuchen beantworten zu können, aber wie wir in Teil 2 sehen werden, ist dies (zumindest in einem recht allgemeinen Modell) für externe Datenstrukuren nicht möglich.

Allerdings gibt es im zweidimensionalen Fall interessante Spezialfälle, bei denen man tatsächlich Datenstrukturen finden kann, die die gleiche asymptotische Platz- und Zeitkomplexität wie ein B-Baum im eindimensionalen Fall haben. In Teil 3 werden wir solch einen Spezialfall betrachten und die ersten Schritte der Konstruktion einer optimalen Datenstruktur nachvollziehen.

Wir werden hier nur den Fall einer einzelnen Festplatte betrachten und wie in [8] mit N die Problemgröße (also $N = |P|$) und mit B die EM-Blöckgröße bezeichnen.

2 Eine untere Speicherplatz-Schranke für 2d Bereichssuchen

Subramanian et al. [7] zeigen im *external memory pointer machine* Modell, daß Methoden zur allgemeinen 2d Bereichssuche, die Anfragen der Größe Z in *worst-case* $O((\log_B N)^c + Z/B)$ I/O's bearbeiten können, $\Omega(\frac{N}{B}\log(\frac{N}{B})/(\log\log_B N))$ EM-Speicherblöcke benötigen.

Im external memory pointer machine Modell wird eine Datenstruktur durch einen Graphen $G = (V, E)$ repräsentiert (Abbildung 1). Jeder Knoten entspricht einem Block des externen Speichers und kann bis zu B Daten- oder Zeigerfelder enthalten. Einer der Knoten wird als *Ausgangsknoten* s gekennzeichnet.

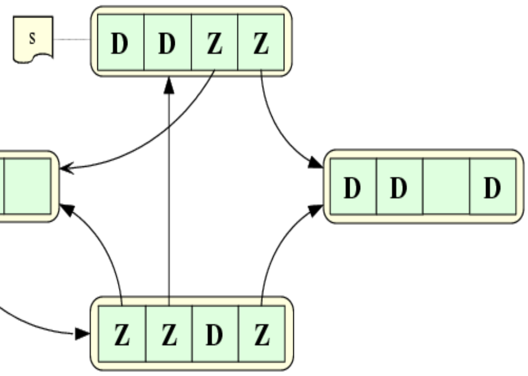

Abbildung 1: *Im "external memory pointer machine" Modell wird eine Datenstruktur durch einen Graph repräsentiert, wobei ein Knoten einen EM-Block darstellen (hier mit $B = 4$).*

Ein Suchalgorithmus muß den folgenden Bedingungen genügen:

- Außer dem Ausgangsknoten s darf ein Knoten nur dann besucht werden, wenn bereits ein Zeiger auf diesen gelesen wurde.
- Für jedes Ausgabedatum muß ein Knoten gelesen werden, der dieses Datum enthält.

Der Suchalgorithmus darf Zeiger zwischen bereits besuchten Knoten hinzufügen, solange in keinem Knoten dadurch mehr als B Felder gespeichert werden. Die Mächtigkeit der Menge der besuchten Knoten wird als Anzahl der benötigten I/O's interpretiert.

In diesem Modell interessieren wir uns für eine untere Schranke für die Größe einer Datenstruktur, die Anfragen der Form $q = \{(x, y) \in P | X_l \leq x \leq X_h, Y_l \leq y \leq Y_h\}$ für feste $a, c \geq 1$ in worst-case $a((\log_B N)^c + Z/B)$ I/O's beantworten kann. Die Idee zum Auffinden dieser Schranke ist, eine möglichst große Menge von Anfragen über einer Punktemenge P zu konstruieren, wobei jede dieser Anfragen möglichst viele Punkte enthält, aber möglichst wenige Punkte mit einer anderen Anfrage gemeinsam hat.

Im weiteren bezeichnen wir mit W die Menge der bei einer Anfrage besuchten Knoten der Datenstruktur. Es muß also gelten $|W| \leq a((\log_B N)^c + Z/B)$.

Betrachten wir nun eine Anfrage q der Größe $Z = B(\log_B N)^c$, dann muß gelten $|W| \leq 2a(\log_B N)^c$. Da jeder Knoten in W höchstens B Punkte enthalten kann, gibt es mindestens $\frac{1}{2}(\log_B N)^c$ Knoten in W mit mindestens $\frac{B}{4a}$ Punkten. Knoten mit dieser Eigenschaft bezeichnen wir als *volle Knoten bezüglich* q. Zwei

Anfragen, die weniger als $\frac{B}{4a}$ Punkte gemeinsam haben, haben folglich keine gemeinsamen vollen Knoten.

Ist $S = \{q_1, ..., q_h\}$ also eine Menge von Anfragen, wobei zu jedem q_i $B(\log_B N)^c$ Punkte gehören und q_i, q_j für $i \neq j$ nur höchsten $\frac{B}{8a}$ Punkte gemeinsam haben, dann gibt es $\frac{1}{2}|S|(\log_B N)^c$ verschiedene volle Knoten, und damit muß der Graph der Datenstruktur mindestens so viele Knoten enthalten.

Um eine Anfragemenge S mit den obigen Eigenschaften zu konstruieren, legen wir zuerst die Punktmenge P fest, über der die Anfragen ausgewertet werden sollen. Wir werden versuchen, P so zu konstruieren, daß das weitere Vorgehen möglichst einfach wird.

Da zwei Anfragen höchsten $\frac{B}{8a}$ Punkte gemeinsam haben dürfen, fassen wir gerade soviele Punkte zusammen, indem wir ihnen den gleichen Wert (x, y) zuteilen (wir gehen davon aus, daß wir trotzdem noch alle Punkte unterscheiden können). Dies vereinfacht unsere Aufgabe, denn nun müssen wir nur darauf achten, daß zwei Anfragen höchstens einen Wert gemeinsam haben. Für unsere Punktmenge P heißt das, daß sie aus N Punkten mit insgesamt $8a\frac{N}{B}$ Werten besteht.

Jede Anfage aus S muß $B(\log_B N)^c$ Punkte, also $\alpha := 8a(\log_B N)^c$ Werte enthalten. Wir setzen $\beta := (\log(8a) + \log(\frac{N}{B}))/(\log(8a) + \log(\log_B N)^c)$, damit $\alpha^\beta = 8a\frac{N}{B}$, und wählen dann N so, daß α und β ganze Zahlen sind. Dadurch können wir jetzt β-stellige Zahlen zur Basis α betrachten, von denen es ja gerade α^β viele gibt, und mit Hilfe dieser unsere Wertemenge als $\{(u_1...u_\beta, u_\beta...u_1) : u_i \in \{0, ..., \alpha - 1\}\}$ definieren.

Jeder Wert kann nach der obigen Konstruktion mit einer Zahl $U = u_1...u_\beta$ zur Basis α identifiziert werden. Hält man nun alle Ziffern einer solchen Zahl bis auf eine Ziffer u_i fest und läßt u_i von 0 bis $\alpha - 1$ laufen, erhält man eine Punktmenge Q der Mächtigkeit α.

Q entspricht nun gerade einer Anfrage der Form $q = \{(x, y) \in P | X_l \leq x \leq X_h, Y_l \leq y \leq Y_h\}$, wobei $X_l := u_1...u_{i-1}0u_{i+1}...u_\beta$ und $X_h := u_1...u_{i-1}(\alpha - 1)u_{i+1}...u_\beta$, sowie $Y_l := u_\beta...u_{i+1}0u_{i-1}...u_1$ und $Y_h := u_\beta...u_{i+1}(\alpha - 1)u_{i-1}...u_1$.

Um dies einzusehen, betrachten wir einen Punkt (x_1, y_1), der Teil von q ist. Die ersten $i - 1$ Ziffern von x_1 bzw. y_1 müssen folglich gerade $u_1...u_{i-1}$ bzw. $u_\beta...u_{i+1}$ entsprechen. Wegen unserer Konstruktion der Punktmenge sind aber die ersten $i - 1$ Ziffern von x_1 gleich den (gespiegelten) letzten $i - 1$ Ziffern von y_1, was analog für die ersten Ziffern von y_1 und den letzten von x_1 gilt. Damit haben wir also $(x_1, y_1) = (u_1...u_{i-1}vu_{i+1}...u_\beta, u_\beta...u_{i+1}vu_{i-1}...u_1)$ für ein $v \in \{1, ..., \alpha - 1\}$ und daraus folgt $(x_1, y_1) \in Q$.

Da wir β Möglichkeiten haben, die variable Ziffer zu wählen, und es dann $\alpha^{\beta-1}$ Möglichkeiten für die Belegung der festen Ziffern gibt, können wir auf diese Weise eine Menge von $\beta\alpha^{\beta-1}$ Anfragen mit je $B(\log_B N)^c$ Punkten generieren.

Nun haben zwei verschiedene dieser Anfragen q_1, q_2 höchstens einen Wert gemeinsam, nämlich dann, wenn alle festen Ziffern von q_1 den entsprechenden festen Ziffern von q_2 entsprechen (wenn man die Anfragen mit der Zahl U identifiziert, mit Hilfe derer sie erzeugt werden).

Die konstruierte Anfragemenge hat also die von uns verlangten Eigenschaften, womit in diesem Fall der Graph der verwendeten Datenstruktur mindestens

$\frac{1}{2}\beta\alpha^{\beta-1}(\log_B N)^c$ Knoten haben muß. Es gilt

$$\beta\alpha^{\beta-1} = \frac{\frac{N}{B}\log(8a)+\log(\frac{N}{B})}{(\log_B N)^c(\log(8a)+\log(\log_B N)^c)} = \Omega(\frac{\frac{N}{B}\log(\frac{N}{B})}{c(\log_B N)^c(\log\log_B N)}).$$

Damit kommen wir zu dem Ergebnis:

Im external memory pointer machine Modell muß eine Datenstruktur G, die allgemeine 2d orthogonale Bereichssuchen der Größe Z über einer Punktemenge P mit $|P| = N$ *für feste* $a, c \geq 1$ *in worst-case* $O((\log_B N)^c + Z/B)$ *I/O's beantworten kann, mindestens* $\Omega(\frac{\frac{N}{B}\log(\frac{N}{B})}{(\log\log_B N)})$ *Knoten enthalten.*

Die Existenz von Datenstrukturen, die allgemeine Anfragen in $O(\log_B N + \frac{Z}{B})$ I/O's beantworten können und $O(\frac{\frac{N}{B}\log(\frac{N}{B})}{(\log\log_B(N)+1)})$ EM-Blöcke benötigen ([8], Seite 51) zeigt, daß die gefundene untere Schranke für den benötigten Speicherplatz eng ist.

3 Effizienter lösbare Spezialfälle der 2d Bereichssuche

Im vorhergehenden Teil haben wir gesehen, daß wir im allgemeinen keine Datenstruktur für die 2d Bereichssuche angeben können, die so effizient ist wie ein B-Baum im 1d Fall. Der zentrale Punkte der Herleitung der unteren Schranke war die Konstruktion einer Menge von Anfragen, die sich möglichst wenig überlappen. Diese Konstruktion sollte nicht mehr gelingen, wenn wir nur 2- oder 3-seitige Anfragen zulassen (Abbildung 2). Tatsächlich gibt es für diese Fälle Datenstrukturen, die asymptotisch die Effizienz des B-Baumes erreichen.

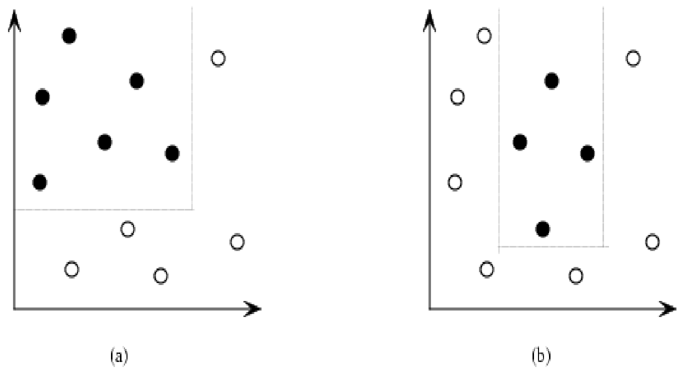

Abbildung 2: *(a) eine 2-seitige Anfrage und (b) eine 3-seitige Anfrage. In beiden Fällen sind die ausgewählten Punkte schwarz gefüllt.*

3.1 2d "diagonal corner" Anfragen

Wir wollen hier einen Spezialfall der 2-seitigen Anfragen genauer betrachten, nämlich die sogenannten *diagonal corner* Anfragen (Abbildung 3): Gegeben ein Punkt $p := (x_p, x_p)$ auf der ersten Winkelhalbierenden, finde alle Punkte (x_r, y_r) mit $0 \leq x_r \leq x_p$ und $x_p \leq y_r$, also alle Punkte die rechts und oberhalb von p liegen. Diese Anfragen sind auch deswegen interessant, weil sie äquivalent sind zu *stabbing* Anfragen, bei denen es darum geht, alle Intervalle der Form $[x_r, y_r]$ zu finden, die einen Punkt x_p enthalten.

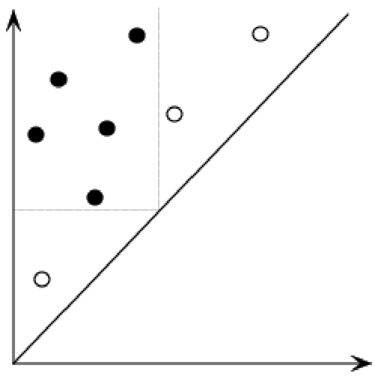

Abbildung 3: *Eine "diagonal corner" Anfrage. Die ausgewählten Punkte sind schwarz gefüllt.*

Zur Vereinfachung machen wir hier folgende Annahmen:

- Die Menge $\mathcal{I}$ der Intervalle, über der wir stabbing Anfragen durchführen, ist bekannt und fest.
- Zwei verschiedene Intervalle haben keinen Endpunkt gemeinsam.

Diese Einschränkungen sind nicht notwending, um eine Datenstruktur mit der gewünschten Effizienz konstruieren zu können [1]. Tatsächlich ist die im folgenden beschriebene Struktur der Ausgangspunkt für eine allgemeinere "dynamische" Datenstruktur ohne Einschränkungen, deren Konstruktion allerdings über den Rahmen dieser Ausarbeitung hinausgeht.

3.1.1 Eine optimale interne Datenstruktur

Für den internen Fall stellt die von Edelsbrunner ([4], hier beschrieben nach [2]) vorgeschlagene Variante des *Intervall Baumes* eine optimale Datenstruktur dar. Betrachten wir eine Menge $\mathcal{I} = \{I_1, I_2, ..., I_N\}$ von Intervallen und die geordnete Folge ihrer verschiedenen Endpunkte $X = (x_1, ..., x_h)$, dann ist ein Intervall Baum T für $\mathcal{I}$ wie folgt definiert (Abbildung 4): Der Wurzel von T wird die Diskriminante $\delta_r = x_r = x_{\lceil \frac{h}{2} \rceil}$ zugeordnet, die $\mathcal{I}$ in drei Teilmengen partitioniert

- $\mathcal{I}_l = \{I_i \in \mathcal{I} | b_i < \delta_r\}$,
- $\mathcal{I}_r = \{I_i \in \mathcal{I} | a_i > \delta_r\}$ und

- $\mathcal{I}_{\delta_r} = \{I_i \in \mathcal{I} | a_i \leq \delta_r \leq b_i\}$.

Die Intervalle in $\mathcal{I}_{\delta_r}$ werden in zwei Listen $\mathcal{AL}$ und $\mathcal{DR}$ gespeichert:

- $\mathcal{AL}$ enthält alle Elemente von $\mathcal{I}_{\delta_r}$ in nicht absteigender Folge der linken Endpunkte.

- $\mathcal{DR}$ enthält alle Elemente von $\mathcal{I}_{\delta_r}$ in nicht aufsteigender Folge der rechten Endpunte.

Der linke Teilbaum wird rekursiv über $\mathcal{I}_l$ und der Endpunktfolge $(x_1, ..., x_{\lceil \frac{h}{2} \rceil - 1})$ definiert, der rechte entsprechend über $\mathcal{I}_r$ und $(x_{\lceil \frac{h}{2} \rceil + 1}, ..., x_u)$.

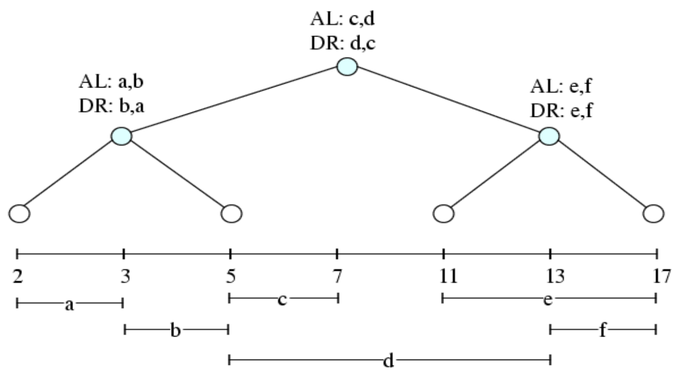

Abbildung 4: *Ein Intervall Baum für 6 Intervalle mit Endpunkten in* $\{2, 3, 5, 7, 11, 13, 17\}$

Um eine stabbing Anfrage bezüglich einem Punkt x zu beantworten, wird zuerst die Wurzel von T durchsucht und dann rekursiv einer der beiden Unterbäume:

- Falls $x < \delta_r$ ist, wird $\mathcal{AL}$ durchlaufen bis ein Intervall gefunden wird, dessen linker Endpunkt größer als x ist, und alle bis dahin gelesenen Intervalle werden ausgegeben; dann wird der linkte Teilbaum rekursiv durchsucht.

- Falls $q > \delta_r$ ist, wird $\mathcal{DR}$ durchlaufen, bis ein Intervall gefunden wird, dessen rechter Endpunkt kleiner als x ist, und alle bis dahin gelesenen Intervalle werden ausgegeben; dann wird der rechte Teilbaum rekursiv durchsucht.

- Falls $q = \delta_r$ ist, wird die gesamte Liste $\mathcal{AL}$ ausgegeben.

Ein Intervall Baum benötigt zum Speichern von N Intervallen $O(N)$ Platz und kann stabbing Anfragen der Größe Z in $O(\log h + Z) = O(\log N + Z)$ Zeiteinheiten bearbeiten (h ist die Anzahl der verschiedenen Intervallendpunkte und damit $h = 2N$).

Wir wollen im folgenden eine externe Version dieses Intervall Baumes entwickeln.

3.1.2 Ein externer Intervall Baum

Die oberen Schranken für die externe Datenstruktur sind:

- Speichern von N Intervallen in $O(\frac{N}{B})$ EM-Blöcken.
- Bearbeiten von stabbing Anfragen der Größe Z in $O(\log_B N + \frac{Z}{B})$ I/O's.

Um stabbing Anfragen in der verlangten Anzahl I/O's ausführen zu können, müssen wir die Höhe des Baumes auf $O(\log_B N)$ reduzieren. Um dies zu Erreichen, erhöhen wir den fan-out auf $\sqrt{B}$ (weshalb wir den fan-out nicht auf B erhöhen wird weiter unten deutlich).

Um die Analyse zu vereinfachen, gehen wir davon aus, daß die Menge E der Intervallendpunkte die Mächtigkeit $|E| = (\sqrt{B})^i B$ für ein $i \geq 0$ besitzt. Dann ist der verwendete Baum perfekt balanciert und jedes Blatt repräsentiert B aufeinanderfolgende Punkte aus E. Wegen der Voraussetzung, daß zwei verschiedene Intervalle keinen gemeinsamen Endpunkt haben dürfen, werden jedem Blatt also höchstens $B/2$ Intervalle zugeordnet, womit wir $O(1)$ EM-Blöcke pro Blatt benötigen.

Die Kinder v_i $(1 \leq i \leq \sqrt{B})$ eines inneren Knotens v partitionieren dessen zugehörige Endpunktfolge in $\sqrt{B}$ sogenannte *slabs* der Form $[b_i, b_{i+1}]$ (Abbildung 5). Mehrere aufeinanderfolgende slabs bezeichnen wir als *multislab*. In v werden die Intervalle gespeichert, die eine slab-Grenze von v aber keine slab-Grenze seines Vaterknotens enthalten. Wir nennen die Menge dieser Intervalle $\mathcal{I}_v$. Für jede der $\sqrt{B}+1$ slab-Grenzen b_i von v speichern wir nun

- eine slab-Liste, die alle Intervalle in $\mathcal{I}_v$ enthält, deren rechter Endpunkt in $[b_{i-1}, b_i]$ liegt, abfallend sortiert nach rechtem Endpunkt;
- eine slab-Liste, die alle Intervalle in $\mathcal{I}_v$ enthält, deren linker Endpunkt in $[b_i, b_{i+1}]$ liegt, aufsteigend sortiert nach linkem Endpunkt;
- eine multislab-Liste für jedes b_j $(j > i)$, die alle Intervalle in $\mathcal{I}_v$ enthält, deren linker Endpunkt in $[b_{i-1}, b_i]$ und deren rechtern Endpunkt in $[b_j, b_{j+1}]$ liegt, abfallend sortiert nach rechtem Endpunkt.

Da es für einen inneren Knoten insgesamt ungefähr $(\sqrt{B})^2/2 = B/2$ multislab-Listen gibt, können wir die Zeiger auf seine sekundären Strukturen in $O(1)$ EM-Blöcken speichern (bei einem fan-out von B würden wir dazu $O(B)$ EM-Blöcke benötigen!).

Betrachten wir nun den benötigten externen Speicher. Wir haben uns eine obere Schranke von $O(\frac{N}{B})$ EM-Blöcken gesetzt und wir können davon ausgehen, daß wir eine Liste mit e Einträgen in $O(\frac{e}{B}+1)$ EM-Blöcken speichern können.

Wir haben gesehen, daß wir für ein Blatt nur $O(1)$ EM-Blöcke benötigen, diese stellen also kein Problem dar.

Uns interessiert jetzt die Anzahl der Strukturen eines inneren Knotens, die im worst-case weniger als $\Theta(B)$ Intervalle enthalten können, deren Speicherverbrauch also nicht durch die enthaltenen Intervalle "bezahlt" werden kann. Dies sind:

- $O(1)$ Verwaltungsdaten,

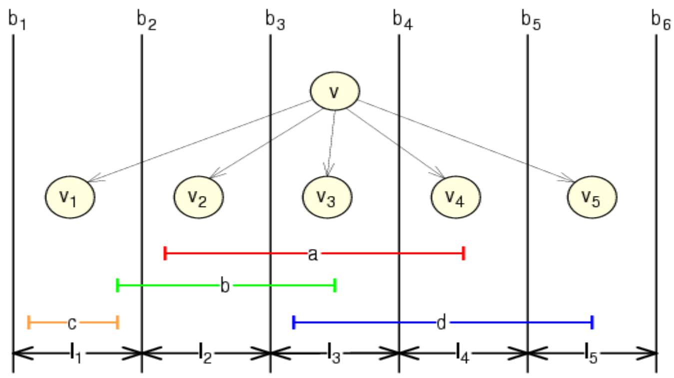

Abbildung 5: *Innerer Knoten eines Intervall Baumes mit $\sqrt{B} = 5$. Intervall a wird in der rechten slab-Liste von b_3 und der linken slab-Liste von b_4, sowie in einer multislab-Liste von b_3 gespeichert.*

- $2\sqrt{B}$ slab-Listen,
- $\frac{B}{2}$ multislab-Listen.

Da es $O(N/(B\sqrt{B}))$ innere Knoten gibt, brauchen wir somit insgesamt bis zu $O(N/B) + O(N/\sqrt{B})$, also $O(N/\sqrt{B})$ EM-Blöcke, aber das ist leider zu viel.

Das Problem liegt bei den $O(B)$ möglicherweise fast leeren multislab-Listen. Um deren Speicherverbauch zu reduzieren, werden multislab-Listen mit weniger als $\Theta(B)$ Intervallen in einer *underflow structure* gespeichert. Genauer werden Listen mit weniger als $B/2$ Elementen in der underflow structure gespeichert. Dadurch ist die Größe der underflow structure garantiert kleiner als B^2, womit diese durch die von Kanellakis et al. [6] entworfene *corner structure* realisiert werden kann. Letztere kann für $k \leq B^2$ Intervalle

- in $O(\frac{k}{B})$ EM-Blöcken repräsentiert werden;
- stabbing Anfragen im worst-case in $O(\frac{Z}{B} + 1)$ I/O's bearbeiten;
- in $O(\frac{k}{B})$ I/O's erstellt werden.

Damit bleiben nur noch höchsten $O(\sqrt{B})$ Strukturen mit weniger als $\Theta(B)$ Intervallen, also brauchen wir ingesamt optimale $O(\frac{N}{B})$ EM-Blöcke zum Speichern des gesamten Intervall Baumes.

Die I/O-Schranke für stabbing Anfragen der Größe Z ist $O(\log_B N + Z/B)$. Wir dürfen folglich nur $O(\log_B N)$ Blöcke mit weniger als $\Theta(B)$ Ausgabeintervallen laden (diese Blöcken nennen wir *leer*).

Angenommen, wir führen eine stabbing Anfrage mit dem Punkt x aus. Dann müssen wir in unserer Datenstruktur alle Knoten auf dem Pfad von der Wurzel bis zu dem Blatt besuchen, das die Intervallendpunkte enthält, zwischen denen sich x befindet.

Sei v ein inneren Knoten auf diesem Pfad und seien b_i und b_{i+1} die slab-Grenzen, zwischen denen x liegt, dann gehen wir wie folgt vor:

- Wir laden die Verwaltungsdaten von v ($O(1)$ leere Blöcke),
- Wir benutzen diese Verwaltungsdaten, um genau die multislab-Listen auszugeben, die b_i und b_{i+1} enthalten (nach Konstruktion keine leeren Blöcke).
- Wir führen eine stabbing Anfrage mit x auf der underflow structure aus und geben das Resultat aus ($O(1)$ leere Blöcke).
- Wir geben alle Intervalle der slab-Liste von b_i aus, die nach rechten Endpunkt sortiert ist, bis ein rechter Endpunkt kleiner als x ist ($O(1)$ leere Blöcke).
- Wir geben alle Intervalle der slab-Liste von b_{i+1} aus, die nach linkem Endpunkt sortiert ist, bis ein linker Endpunkt größer als x ist ($O(1)$ leere Blöcke).

Wir laden also tatsächlich höchstens $O(\log_B N)$ leere Blöcke und haben damit unter den oben genannten Einschränkungen gezeigt:

Es gibt eine Datenstruktur, die N Intervalle in $O(N/B)$ EM-Blöcken speichern kann, wobei keine zwei Intervalle einen gemeinsamen Endpunkt haben, und die stabbing Anfragen der Größe Z in worst-case $O(\log_B N + Z)$ I/O's bearbeiten kann.

3.1.3 Ein dynamischer EM Intervall Baum

Erlaubt man in der oben vorgestellten Datenstruktur das Einfügen und Löschen von Intervallen, muß man dafür sorgen, daß der Baum balanciert bleibt, um eine Suchpfadlänge in $O(\log_B N)$ zu garantieren. Das Balancieren kann in wie im Fall des B-Baumes in worst-case $O(\log_B N)$ I/O's durchgeführt werden [1], allerdings ist dies wegen der sekundären Strukturen jedes inneren Knotens, die von der Position des Knotens im Baum abhängen, im Vergleich zum bekannten Vorgehen beim B-Baum [3] wesentlich aufwendinger.

Literatur

[1] L. Arge and J.S. Vitter. Optimal dynamic interval management in external memory (extended abstract). In *IEEE Symposium on Foundations of Computer Science*, pages 560–569, 1996.

[2] P. Cignoni, P. Marino, C. Montani, E. Puppo, and R. Scopigno. Speeding Up Isosurface Extraction Using Interval Trees. *IEEE Transactions on Visualization and Computer Graphics*, 3(2):158–170, 1997.

[3] T.H. Cormen, C.E. Leiserson, and R.L. Rivest. *Introduction to Algorithms*. M.I.T. Press, Cambridge, Massachusetts, U.S.A., 1990. Chapter 19.

[4] H. Edelsbrunner. Dynamic data structures for orthogonal intersection queries. Technical report, Inst. Informationsverarb., Tech. Univ. Graz, Graz, Austria, 1980.

[5] V. Gaede and O. Gunther. Multidimensional access methods. *ACM Computing Surveys*, 30(2):170–231, 1998.

[6] P.C. Kannellakis, S. Ramaswamy, D.E. Vengroff, and J.S. Vitter. Indexing for Data Models with Constraints and Classes. In *Proc. ACM Symp. Principles of Database Systems*, pages 233–243, 1993.

[7] S. Subramanian and S. Ramaswamy. The P-range tree: A new data structure for range searching in secondary memory. In *6 th ACM-SIAM Symposium on Discrete Algorithms*, pages 378–387, 1995.

[8] J.S. Vitter. External Memory Algorithms and Data Structures: Dealing with MASSIVE DATA. 2001. available from the authors homepage: `http://www.cs.duke.edu/~jsv`.