Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

Eines der wichtigsten Gebiete der modernen Informationsverarbeitung besteht sicherlich in der Sicherung und Verwaltung anfallender Daten. Die Notwendigkeit, gesammelte Daten zu speichern und auf diese für den späteren Gebrauch effizient zugreifen zu können, wurde durch die sich stark vergrößernden Datenbestände immer dringlicher. Heute werden in nahezu jedem Einsatzgebiet der Informationstechnologie große Mengen Daten unterschiedlichster Art erzeugt. Längst reicht es nicht mehr aus, diese Daten unabhängig von ihrer Struktur mit allgemein verwendbaren Speichermechanismen zu sichern. Die Diversität der anfallenden Daten der modernen Informationsverarbeitung machte es statt dessen nötig, Speicherstrukturen zu entwerfen, die sich für eine bestimmte Teilmenge der heterogenen Gesamtheit aller anfallender Daten besonders eignen. Durch diese Spezialisierung auf eine Untermenge aller verschiedener Daten lassen sich dann für dieses Teilgebiet besonders wichtige Operationen auf den Datenbeständen effizienter und einfacher realisieren. Eine dieser Teilmengen ist die Menge der mehrdimensionalen Daten, wiederum eine Untermenge davon ist die Menge der räumlichen Daten (engl. Spatial Data), bei denen die Anzahl von Dimensionen bei zwei, z.B. geografische Länge und Breite, oder drei, z.B. geografische Länge, Breite und Höhe, liegt.

Die Menge der räumlichen Daten erlangte mit der fortschreitenden Entwicklung sogenannter geografischer Informationssysteme (engl. Geographic Information Systems, kurz: GIS) eine zunehmende Bedeutung in der Informationsverarbeitung [OGI01]. Der Begriff eines geografischen Informationssystems wurde geprägt durch dessen Anwendungsgebiet, nämlich im weitesten Sinne die Speicherung und Auswertung von Daten in einem geografischen Kontext, wie z.B. die Beantwortung der Frage, welche Städte beim Abfahren einer gewissen Route passiert werden, oder welche Kunden eines Unternehmens in diesen Städten leben.

Den Kern jedes geografischen Informationssystemes bildet eine Komponente, die Daten mit geografischem Kontext abspeichern kann, und die so beschaffen ist, daß die für ein GIS typischen Anfragen effizient und einfach beantwortet werden können. Typische Anfragen an ein GIS wären z.B. die sogenannte Positionsanfrage (engl. Position Query), die bei Eingabe einer geografischen Position die Menge der sich an dieser Stelle befindlichen gespeicherten Objekte zurückgibt. In diesem Zusammenhang sind die sogenannten Lokationsbasierten Dienste (engl. Location Based Services) ein Begriff, die diese Techniken nutzen, um

abhängig von der Position eines Klienten diesem jeweils geeignete Dienste anzubieten. Beispiele wären z.B. das Herausfinden des nächsten chinesischen Restaurants, oder bei Wahl einer Notrufnummer auf einem mobilen Telefon die automatische Positionsübermittlung an die entsprechenden Behörden.

In dieser Arbeit sollte im Rahmen des von den Abteilungen Verteilte Systeme und Anwendersysteme des IPVR (Institut für Parallele und Verteilte Höchstleistungsrechner) der Fakultät Informatik der Universität Stuttgart entwickelten NEXUS-Projekts [NEX99] für den dort verwendeten Lokationsdienst eine ebensolche Komponente zur Verwaltung der Positionsinformationen entworfen und prototypisch implementiert werden. Mögliche Alternativen einer solchen Komponente waren eine rein hauptspeicherbasierte Variante, die alle Daten im Hauptspeicher hält, oder eine auf einer herkömmlichen Datenbank basierende Lösung.

Dazu mußten zunächst die funktionalen Anforderungen, die der Lokationsdienst an eine solche Komponente stellt, erarbeitet werden. Diese unterscheiden sich von den allgemeinen Anforderungen, die an ein GIS typischerweise gestellt werden. Gründe hierfür sind unter anderem, daß beim Lokationsdienst der Einsatz von Sekundärspeicher bespielsweise nicht von Wichtigkeit ist. Außerdem sind beim Lokationsdienst durch hohe Anforderungen an die Positionsgenauigkeit auch große Anforderungen an das effiziente Ausführen von Aktualisierungen der Positionsinformationen mobiler Objekte gegeben. Dies führte zur Eignungsprüfung von verschiedenen bekannten Strukturen (sog. Indexstrukturen) zur Verwaltung räumlicher Daten im Hinblick auf die gefundenen Anforderungen. Eine räumliche Indexstruktur bildet den zentralen Punkt einer räumlichen Datenhaltungskomponente. Die Leistungsfähigkeit dieser Struktur bestimmt ganz entscheidend die Leistungsfähigkeit der gesamten Komponente. Weiterhin gibt es keine „beste“ Indexstruktur, die sich für alle Anwendungen gleich gut eignet [GAE98], so daß die Wahl einer solchen immer von der jeweiligen Umgebung abhängig ist.

Nach Durchführen von Tests mit den im Zuge der Forschungsarbeiten implementierten Indexstrukturen sollte auch ein Vergleichstest mit einer kommerziellen Lösung zur Verwaltung räumlicher Daten vorgenommen werden (Spatial Extender für IBM DB2).

Dies alles führte schließlich zur prototypischen Entwicklung einer Speicherkomponente für räumliche Daten für den oben beschriebenen Lokationsdienst. Als Implementierungssprache war hierfür die Programmiersprache JAVA (in der Version 1.3) vorgegeben.

In Kapitel 2 dieser Ausarbeitung wird noch einmal ausführlicher auf die folgenden Themen eingegangen: Eigenschaften räumlicher Daten, die wesentlichen Unterschiede, die zwischen herkömmlichen eindimensionalen und mehrdimensionalen Daten bestehen, die wichtigsten Operationen auf räumlichen Daten und die Vorstellung einer Auswahl von wichtigen Indexstrukturen sowie die Einordnung dieser in verschiedene Klassen von räumlichen Indexstrukturen. Auf die oben bereits erwähnten funktionalen Anforderungen an eine Komponente zur Speicherung der räumlichen Informationen innerhalb eines Servers des Lokationsdienstes wird, zusammen mit einer kurzen Beschreibung dieses Dienstes, in Kapitel 3 eingegangen werden. In Kapitel 4 werden die in Kapitel 2 vorgestellten räumlichen Indexstrukturen auf ihre Eignung im Hinblick auf die in Kapitel 3 zusammengestellten Anforderungen verglichen werden. Diese theoretische Betrachtung wird durch die Beschreibung der mit diesen Strukturen durchgeführten Experimente in Kapitel 5 noch ergänzt. Ebenfalls wird in diesem Kapitel die Durchführung und die gefundenen Resultate von Tests mit einer kommerziellen Lösung (IBM Spatial Extender für DB2 UDB) dargestellt. Eine Diskussion der in den Kapiteln 3 bis 5 beschriebenen Untersuchungen wird in Kapitel 6 angestellt, bevor abschließend in Kapitel 7 noch auf zukünftig mögliche Erweiterungen und Verbesserungen der erstellten Komponente eingegangen wird.

2 Grundlagen räumlicher Daten

2.1 Eigenschaften

Der fundamentale Verwendungszweck räumlicher Daten ist die Abbildung von geografischen Informationen beliebiger Art in eine für digitale Informationsverarbeitungssysteme verständliche, effektiv verwaltbare Form. Nachdem diese Abbildung durchgeführt wurde, lassen sich, wie bei vielen anderen von Informationssystemen verwalteten Datentypen, effizient ausführbare Operationen auf diesen Datenbeständen definieren und durchführen. Geografische Informationen bestehen hierbei ganz allgemein aus der Beschreibung der Objekte, die zusammengesetzt die Erdoberfläche bilden bzw. diese besiedeln (im folgenden daher als "geografische Objekte" bezeichnet) und der Beziehungen dieser Objekte untereinander. Zum einen sind dies Informationen über natürliche geografische Objekte, wie z.B. Flüsse, Seen, Wälder oder Berge, aber auch Informationen über von Menschen geschaffene, künstlich-kulturelle Objekte wie z.B. Städte, Häuser, Straßen, aber auch Theater, Restaurants und deren Beziehungen untereinander.

Reale geografische Objekte lassen sich im zweidimensionalen Raum typischerweise auf einen der folgenden geometrischen Datentypen abbilden:

- *Den Punkt*. Hierauf wird üblicherweise jedes geografische Objekt ohne räumliche Ausdehnung abgebildet, z.B. eine herkömmliche Positionsangabe.
- *Den offenen Linienzug*. Dieser dient zur Darstellung von größeren zusammenhängenden Objekten ohne zweidimensionale Ausdehnung, also z.B. Straßen oder Flüssen (Annahme: Fläche der Straße/des Flusses gleich null).
- *Das Polygon (geschlossener Linienzug)*. Diese geometrische Figur wird zur Abbildung aller geografischer Objekte verwendet, die eine zweidimensionale Ausdehnung besitzen. Es existieren häufig noch Unterklassen des Polygons, die jeweils besonders gut geeignet sind, bestimmte geografische Objekte abzubilden. Zum Beispiel könnte man einen quadratischen Bauplatz auf die Unterklasse "Rechteck", einen runden Platz auf die Unterklasse "Kreis", oder aber einen natürlichen See mit seiner unregelmäßigen Form auf die Oberklasse "Polygon" abbilden.

Wichtige Beziehungen geografischer Objekte untereinander sind beispielsweise:

- Die Lage von geografischen Objekten im Bezug auf ihre Umgebung, z.B. die Lage von Polizeistationen oder Krankenhäusern innerhalb einer Stadt, oder die Nähe einer Stadt zu einem bekannten Erdbebengebiet.
- Die Art und Weise, wie geografische Objekte miteinander in Beziehung stehen, beispielsweise die Information, daß ein bestimmter See komplett in einem bestimmten Land liegt, oder daß ein Wald die Gemarkungsgrenze zweier Gemeinden schneidet.
- Maßzahlen, die sich auf auf ein oder mehrere geografische Objekte beziehen, beispielsweise die geografische Entfernung eines Motels von einer Autobahnausfahrt oder die Länge eines Wanderweges durch die Alpen.

Durch das Wissen über die Eigenschaften geografischer Objekte und deren Beziehungen zu ihrer Umgebung ergibt sich eine Vielzahl von Anwendungsmöglichkeiten, die vom bloßen Feststellen der Fläche eines Bauplatzes bis hin zur Lösung komplexer Fragestellungen wie z.B. "Anzahl der Brücken, die innerhalb eines bestimmten Gebietes einen Fluß überqueren" reichen.

Die oben angedeuteten vielfältigen Möglichkeiten die sich einem Anwender oder einem Unternehmen durch die rechnergestützte Verwaltung und Verwendung von geografischen Informationen bieten, führte in den letzten Jahren zu einer rasanten Entwicklung der GIS, deren Ende zum jetzigen Zeitpunkt noch nicht abzusehen ist.

2.2 Vergleich mit eindimensionalen Daten

2.2.1 Eindimensionales Datenmodell

In den Anfängen der Informationsverarbeitung war eine Unterteilung zwischen eindimensionalen und mehrdimensionalen Daten weitgehend unbekannt. Alle Datensätze die gesammelt wurden, ließen sich gewöhnlich bezüglich eines besonderen Datentyps, des sogenannten Schlüssels vergleichen und damit gemäß einer auf diesem Datentyp definierten Totalordnung sortieren. Beispiele hierfür sind z.B.:

- **Datentyp**: Zahl (ganze Zahl, Bruchzahl, irreguläre Zahl)
  Ordnung: Totalordnung über der entsprechenden Zahlenmenge
- **Datentyp**: Zeichenkette, Zeichen
  Ordnung: lexikografische Totalordnung

Herkömmliche Datenhaltungssysteme wie z.B. die bekannten relationalen Datenbanksysteme (RDBMS) basieren auf dem eindimensionalen Datenmodell.

Zur effizienten Informationssuche können auf dem Schlüsselwert sogenannte "Indizes" definiert werden, vergleichbar mit der Inhaltsangabe eines wissenschaftlichen Buches in Kapitel und Inhalt eines Kapitels. Dies hat den Vorteil, daß die Menge aller Datensätze (analog: das Buch) nicht sequentiell nach einem bestimmten Schlüsselwert durchsucht werden muß, sondern schnell an eine bestimmte Stelle (im Bsp.: ein Kapitel) gesprungen werden kann und die dort verwahrten Daten entnommen werden können. Es existiert eine Vielzahl von Vorschlägen zur Organisation solcher Indizes, diese sind durch die vor allem im Bereich der Datenbanken in der Vergangenheit angestellten Untersuchungen gut erforscht. Bekannte Vertreter dieser Klasse sind z.B. der B-Baum [BMC72] oder das Verfahren des Extensible Hashing [FAG79].

2.2.2 Mehrdimensionale (räumliche Daten)

Mit fortschreitender Entwicklung der Informationsverarbeitung und deren Ausdehnung auf ganz unterschiedliche Anwendungsgebiete zeigte sich, daß das eindimensionale Datenmodell für viele der neuen Anwendungsgebiete nicht mehr ausreichend war. Es mußten nun Datensätze verwaltet werden, deren einzelne Datentypen nicht mehr eindimensional, wie z.B. die natürlichen Zahlen waren, sondern mehrdimensional, wie z.B. der Datentyp "Positionsangabe", der im dreidimensionalen Raum aus einem Tupel mit mindestens drei Werten besteht.

Der Anwendungsbereich der geografischen Informationssysteme ist allerdings nur eine Ausprägung des mehrdimensionalen Datenmodells, andere wichtige Gebiete sind auch grafische Anwendungen wie z.B. der Bereich der Bildverarbeitung. Im folgenden soll jedoch der Bereich der "räumlichen" Daten näher betrachtet werden, der Begriff "mehrdimensional" wird deshalb, wenn nicht anders angegeben, als Synonym für den Begriff "räumlich" verwendet.

Der wesentliche Unterschied zwischen eindimensionalen und räumlichen Daten ist der, daß im Gegensatz zu den eindimensionalen Daten keine Totalordnung existiert, bzw. sich definieren läßt, die die räumliche Nähe der verschiedenen Wertetupel (der Koordinaten eines Punktes) vollständig bewahrt. Anders ausgedrückt, existiert keine allgemeine Abbildungsvorschrift aus einem mehrdimensionalen in einen eindimensionalen Wertebereich, so daß zwei beliebige Mitglieder einer Menge von Objekten, die im mehrdimensionalen Raum nahe beieinander liegen, dies auch im eindimensionalen Raum tun.

Es wurden im Verlaufe der Forschung auf diesem Gebiet bereits einige sogenannte "raumfüllende Kurven" vorgeschlagen, einige davon sind in untenstehender Abbildung skizziert. Gemeinsam haben aber all diese Vorschläge, daß sie keinen vollständigen Erhalt der Topologie bieten können, sondern sich diesem nur in unterschiedlichem Maße annähern. Mit anderen Worten besteht je nach Qualität einer raumfüllenden Kurve eine bestimmte, jedoch hohe Wahrscheinlichkeit, daß zwei Punkte, die in der Realität benachbart sind, dies auch auch nach der Abbildung durch eine solche Funktion sind.

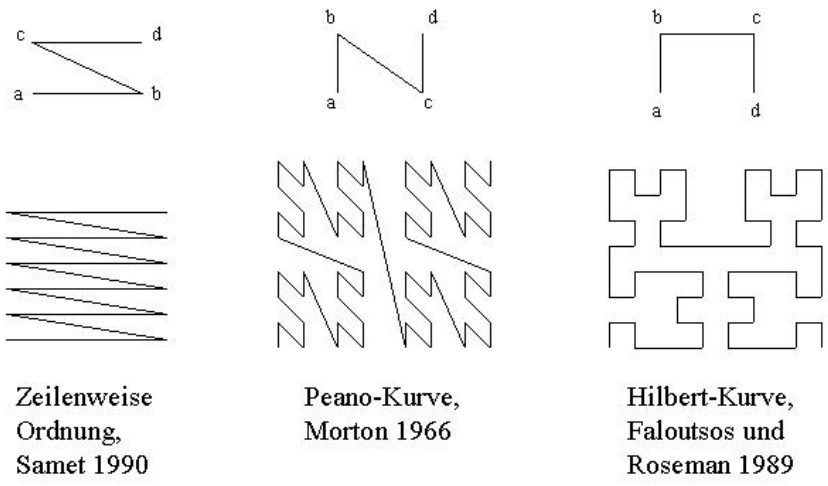


Abbildung 1: Einige Beispiele raumfüllender Kurven

Wie bereits in Unterkapitel 2.2.1 erwähnt, werden Indizes zum effizienten Zugriff auf die aufgezeichneten Daten verwendet. Durch das Nichtvorhandensein einer Totalordnung lassen sich die ebenfalls erwähnten, herkömmlichen Indexstrukturen jedoch nicht effizient für die Verwaltung von räumlichen Daten einsetzen. Theoretisch ist es zwar möglich, wie in der Vergangenheit oftmals angewendet, mehrdimensionale Datentypen in ihre eindimensionalen

Bestandteile aufzuteilen und je einen Index auf jeden dieser Bestandteile zu definieren. Dies führt in der Realität jedoch zu für die meisten Anwendungen nicht tolerierbaren Geschwindigkeitsverlusten beim Zugriff auf diese Daten [KRI84]. Dadurch wurde das Bedürfnis nach Strukturen formuliert, die sich für die Verwaltung von mehrdimensionalen Daten eignen. Bis zur Gegenwart wurden dann auch eine Vielzahl von Indexstrukturen von verschiedener Seite vorgeschlagen, eine Auswahl dieser wird in Kapitel 2.4 vorgestellt werden.

2.3 Typische Operationen

Durch die Verwendung räumlicher Daten in der digitalen Informationsverarbeitung ergab sich für Anwender die Möglichkeit, Fragestellungen, die bislang durch zeitraubende manuelle Vorgänge beantwortet werden mußten, wie z.B. die Antwort auf die Frage: "Wie groß ist die Entfernung zur nächsten Berghütte?", automatisiert und damit schnell zu klären. Dies führte zur Formulierung einiger einfacher Standardanfragen, die jedes GIS bieten sollte.
Im wesentlichen sind dies:

- Übereinstimmens- oder Positionsanfrage (engl:. Exact Match/Position Query).
- Bereichsanfrage (engl.: Range Query).
- Nächster-Nachbar-Anfrage (engl.: Nearest Neighbour Query).
- Schnittanfrage (engl.: Intersection Query).
- Enthaltensanfrage(engl.: Containment Query).
- Enthaltenseinsanfrage (engl.: Enclosure Query).

Die verschiedenen Anfragetypen sollen im folgenden kurz beschrieben werden. Gemeinsam haben all diese Anfragen, daß der Rückgabewert aus einer Menge von Lösungen für diese Anfrage besteht. Ein n-dimensionales Universum *U* ist hierbei als die Menge aller Punkte zu verstehen, die mit *n* Koordinaten beschreibbar sind. Eine wohldefinierte Positionsangabe besteht in einem n-dimensionalen Universum dabei aus einem n-stelliges Tupel $(d_1,d_2,...,d_n)$, ein Fenster aus einem zweistelligen Tupel, bestehend aus einer Menge von Positionsangaben $\{p_1, ..,p_k\}$ und einer einen Zyklus bildenden Menge von Kanten $\{(p_1, p_2), (p_2, p_3), ..., (p_k, p_1)\}$ für ein gewisses *k*. Weiterhin wird für die Kanten eines Fensters gefordert, daß diese isoorientiert sind, d.h. jeweils parallel zu einer das Universum aufspannenden Koordinatenachse sind. Ein (räumliches) Objekt *s* soll im folgenden ein Tupel (σ, id) sein,

wobei σ die räumliche Ausdehnung und *id* ein eindeutiger Bezeichner für dieses Objekt sein soll, die Menge aller räumlicher Objekte *s* soll mit *S* bezeichnet werden. Besitzt ein räumliches Objekt keine Ausdehnung (Punkt), so bezeichnet σ lediglich die Position des Punktes, ist also selbst eine Positionsangabe.

- **Übereinstimmungsanfrage/Positionsanfrage**

  Eingabe: Die räumliche Ausdehnung eines Objektes σ

  Ausgabe: Die Menge der Objektbezeichner der Objekte aus *S*, deren Ausdehnung mit σ übereinstimmt.

  Lösungsmenge: $L = \forall(\sigma_i, id) \in U : \{id \mid ((\sigma_i, id) \in S) \wedge (\sigma_i = \sigma)\}$

Diese Anfrage dient zur Bestimmung der Objekte, deren räumliche Ausdehnung zu der des Suchobjektes identisch ist, bzw. im Falle eines Punktes dieselbe Position darstellt.

- **Bereichsanfrage**

  Eingabe: ein Suchfenster *W*

  Ausgabe: Die Menge der Objektbezeichner der Objekte aus *S*, die mindestens einen Punkt mit *W* gemeinsam haben.

  Lösungsmenge: $L = \forall(\sigma_i, id) \in U : \{id \mid ((\sigma_i, id) \in S) \wedge ((W \cap \sigma_i) \neq \varnothing)\}$

  Diese Anfrage dient dazu, all die Objekte zu bestimmen, die sich in einem bestimmten Gebiet befinden.

- **Nächster-Nachbar-Anfrage**

  Eingabe: Die räumliche Ausdehnung eines Objektes σ, eine obere Schranke für den Suchbereich *d*.

  Ausgabe: Der Objektbezeichner des Objektes aus *S*, dessen Entfernung zum Suchpunkt die kleinste aller Objekte aus *S* ist. Weiterhin muß diese Entfernung kleiner als *d* sein.

  Lösungsmenge:

  $L = \{id_l \mid ((\sigma_l, id_l) \in S) \wedge (dist(\sigma_l, \sigma) < d) \wedge (\forall k \neq l : (dist(\sigma_l, \sigma) < dist(\sigma_k, \sigma))\}$

Diese Anfrage dient dazu, das zu einem Objekt nächstgelegene Objekt zu bestimmen. Die Spezifikation der oberen Schranke vermeidet in der Praxis nicht verwendbare Ergebnisse.

- **Schnittanfrage**

  Eingabe: Die räumliche Ausdehnung eines Objektes σ.

  Ausgabe: Die Menge der Objektbezeichner der Objekte aus *S*, die mindestens einen Punkt mit σ gemeinsam haben.

  Lösungsmenge: $L = \forall(\sigma_i, id) \in U : \{id \mid ((\sigma_i, id) \in S) \wedge ((\sigma \cap \sigma_i) \neq \emptyset)\}$

  Diese Anfrage dient dazu, all die Objekte zu bestimmen, die mit einem bestimmten Objekt überlappen. Dies ist eine allgemeinere Form der Bereichsanfrage.

- **Enthaltensanfrage**

  Eingabe: Die räumliche Ausdehnung eines Objektes σ.

  Ausgabe: Die Menge der Objektbezeichner der Objekte aus *S*, die vollständig in σ enthalten sind.

  Lösungsmenge: $L = \forall(\sigma_i, id) \in U : \{id \mid ((\sigma_i, id) \in S) \wedge ((\sigma \cap \sigma_i) = \sigma)\}$

  Mit Hilfe dieser Anfrage lassen sich für ein bestimmtes Objekt alle von diesem umschlossenen Objekte bestimmen.

- **Enthaltenseinsanfrage**

  Eingabe: Die räumliche Ausdehnung eines Objektes σ.

  Ausgabe: Die Menge der Objektbezeichner der Objekte aus *S*, die σ vollständig enthalten.

  Lösungsmenge: $L = \forall(\sigma_i, id) \in U : \{id \mid ((\sigma_i, id) \in S) \wedge ((\sigma \cap \sigma_i) = \sigma)\}$

  Mit Hilfe dieser Anfrage lassen sich alle Objekte, die ein bestimmtes Objekt vollständig umschließen, bestimmen. Dies ist die inverse Operation zur Enthaltensanfrage (s.o.).

Diese Anfragen bilden eine Operationsbasis, mit deren Kombination sich nach Bedarf komplexere Anfragen definieren lassen, z.B. "Welche ist die nächste Tankstelle, die sich in einem bestimmten Nachbarland befindet?" als Kombination aus Enthaltensanfrage und Nächster-Nachbar-Anfrage.

2.4 Räumliche Indexstrukturen

Die in diesem Kapitel bereits angesprochenen Indexstrukturen für mehrdimensionale Daten lassen sich nach ihrem hauptsächlichen Einsatzgebiet grob unterteilen in

- Hauptspeicher-basierte räumliche Indexstrukturen,
- Indexstrukturen zum Zugriff auf räumliche Punkte (engl.: Point Access Methods, kurz: PAMs),
- Indexstrukturen zum Zugriff auf ausgedehnte räumliche Gebilde (engl.: Spatial Access Methods, kurz SAMs).

Wichtige Vertreter aus jeder dieser drei Klassen werden nun in den folgenden Unterkapiteln vorgestellt werden. Bildbeispiele beziehen sich dabei jeweils auf die in folgender Abbildung dargestellte Verteilung von Punkten im Raum.

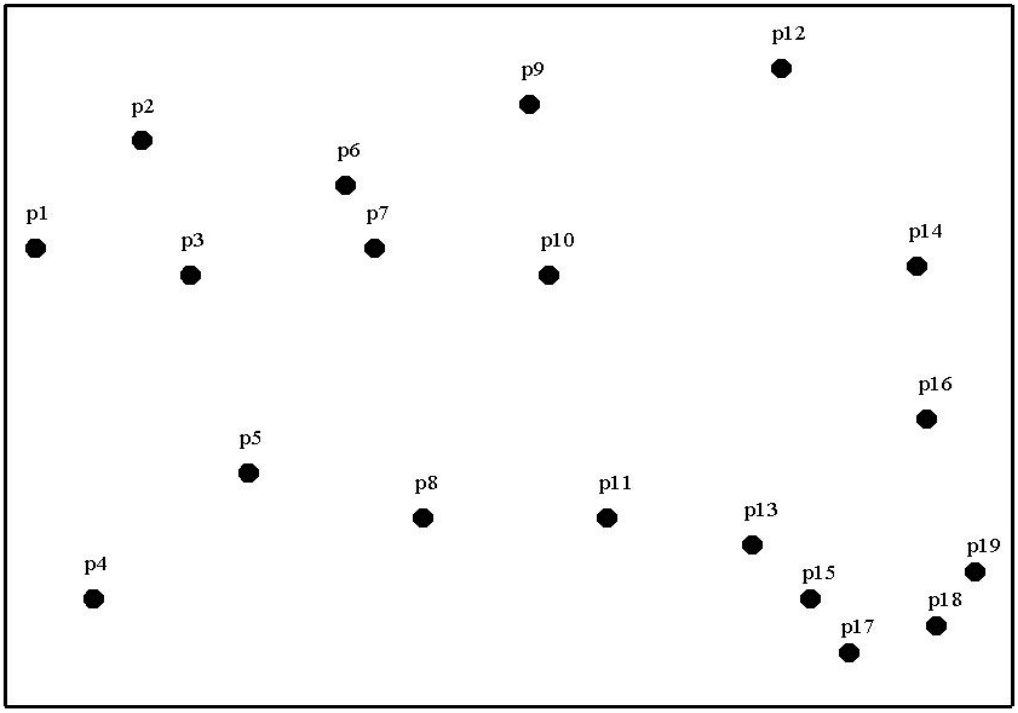


Abbildung 2: Den nachfolgenden Beispielen zugrunde gelegte Objektverteilung

Die Vor- und Nachteile der verschiedenen Indexstrukturen im Bezug auf den praktischen Einsatz im Umfeld des Lokationsdienstes der NEXUS-Plattform werden dann in den Kapiteln 4 und 5 besprochen. Zunächst soll allerdings in einem eigenen Unterkapitel auf wünschenswerte Eigenschaften, die räumliche Indexstrukturen unabhängig von ihrer Klasse besitzen sollten, und die sich aus der Natur geografischer Daten ergeben, eingegangen werden.

Für die räumlichen Indexstrukturen „kd-Baum“, „Quad-Tree“ und „R-Baum“ und einige andere, hier nicht im Detail behandelte Indexstrukturen, existiert eine hervorragende von Frantisek Brabec und Hanan Samet zur Verfügung gestellte interaktive Anwendungsdemonstration (http://www.cs.umd.edu/~brabec/Quad-Tree/). Hier kann der dynamische Aufbau einer der drei Indexstrukturen durch wiederholtes Einfügen von räumlichen Objekten, das Restrukturierungsverhalten bei Löschen einiger räumlicher Objekte und das Suchvorgehen bei Bearbeitung von Anfragen z.B. der Typen „Nächster-Nachbar-Anfrage“ und „Bereichsanfrage“ anhand einer grafischen Darstellung der jeweiligen Struktur verfolgt werden. Gerade die grafische Darstellung der internen Abläufe, z.B. die von einer „Nächster-Nachbar-Anfrage“ vorgenommene Abfolge von Suchschritten in verschiedenen Knoten veranschaulicht die zunächst oft nur abstrakt wahrgenommene Vorgehensweise zur Beantwortung dieser Anfragen.

2.4.1 Allgemeine Anforderungen

Grundprobleme, an deren Behandlung sich jede mehrdimensionale Indexstruktur messen lassen muß und von deren Beachtung unter anderem die effiziente Beantwortung der in 2.3 vorgestellten Anfragen abhängt, sind nach [HÄR99]:

- Die Erhaltung der topologischen Struktur der zu verwaltenden räumlichen Objekte in der Indexstruktur.
- Die Maßnahmen, die eine räumliche Indexstruktur zur Verwaltung von Objekten mit stark variierender räumlicher Dichte ergreift.
- Die Durchführung dynamischer Reorganisationsmaßnahmen.

Der erste Punkt, *Erhaltung der topologischen Struktur*, erhält seine Wichtigkeit aus der typischen Lokalität der Datenzugriffe. Dies hängt direkt mit der Formulierung der Anfragen zusammen, die häufig Antworten auf Fragestellungen wie „alle Häuser einer Straße“, oder

„alle Brücken über einen Fluß in einem bestimmten Gebiet“ liefern sollen. Wenn nun Objekte, die in der Realität nahe beisammen liegen, auch in denselben Blättern, oder in derselben Zelle in einer mehrdimensionalen Indexstruktur gehalten werden, so erleichtert dies die Zusammenstellung der Lösungsmenge erheblich, denn viele Kandidaten der Lösungsmenge können bereits durch Auffinden eines einzigen Blattes oder einer Zelle identifiziert werden. Vor allem Indexstrukturen, die die Verwaltung des Sekundärspeicherspeichers mit einschließen, wie z.B. die in der Klasse der PAMs und der SAMs enthaltenen Indexstrukturen, profitieren von dieser Eigenschaft. Teure Vorgänge sind hier die Positionierung des Schreib/Lesekopfes auf die richtige Plattenposition. Bei einer vollständigen Erhaltung der topologischen Struktur wird die Anzahl dieser Vorgänge minimiert.

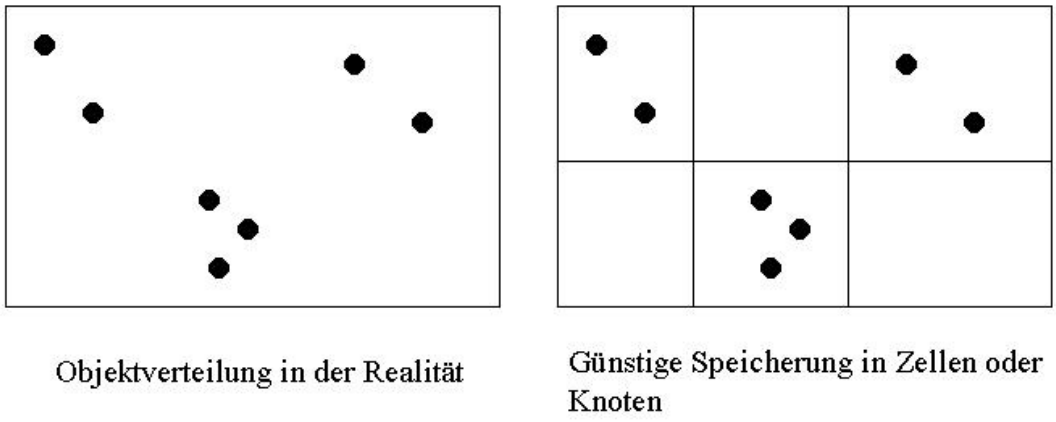


Abbildung 3: Erhaltung der topologischen Struktur

Der zweite Punkt, *Anpassungsfähigkeit an variierende Objektdichten*, soll die Fähigkeit einer Indexstruktur beschreiben, die Partitionierung des zur Verfügung stehenden Raumes so vorzunehmen, daß auch bei stark ungleichmäßiger Verteilung jede Zelle oder Baumknoten mit mindestens x, aber höchstens y (für gewisse Werte x und y) Objekten besetzt ist. Die untere Grenze dient der Durchsetzung eines Mindestmaßes an Speicherausnutzung, die obere Grenze ergibt sich aus dem physischen Fassungsvermögen einer Zelle oder eines Knotens. [FRE97] gibt an, daß bei geografischen Anwendungen die Verteilung der Objekte im Verhältnis $1:10^4$ variieren kann. Hieraus läßt sich ableiten, daß eine Indexstruktur zur Vermeidung einer schlechten Speicherausnutzung ein großes Maß an Anpassungsfähigkeit mitbringen sollte.

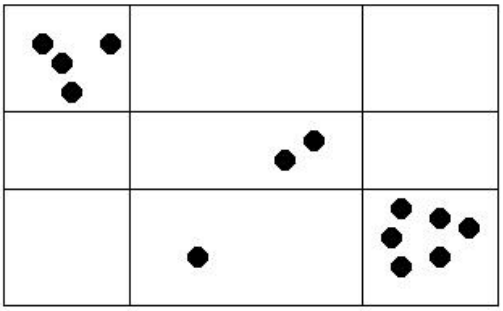


Abbildung 4: Variierende Objektverteilung

Der dritte und letzte hier besprochene Punkt, *Durchführung dynamischer Reorganisationsmaßnahmen*, beschreibt die Fähigkeit einer Indexstruktur, auch in einer dynamischen Umgebung, die sich durch häufiges, unvorhersehbares Durchführen von Einfüge- und Löschvorgängen charakterisieren läßt, eine derart gestaltete Struktur zu behalten, so daß eine vernünftige Speicherausnutzung und ein zufriedenstellendes Antwortverhalten auf gestellte Anfragen gegeben ist. Dies kann etwa durch die angesprochenen dynamischen Reorganisationsmaßnahmen erreicht werden, die bei drohender Entartung einer Indexstruktur durchgeführt werden.

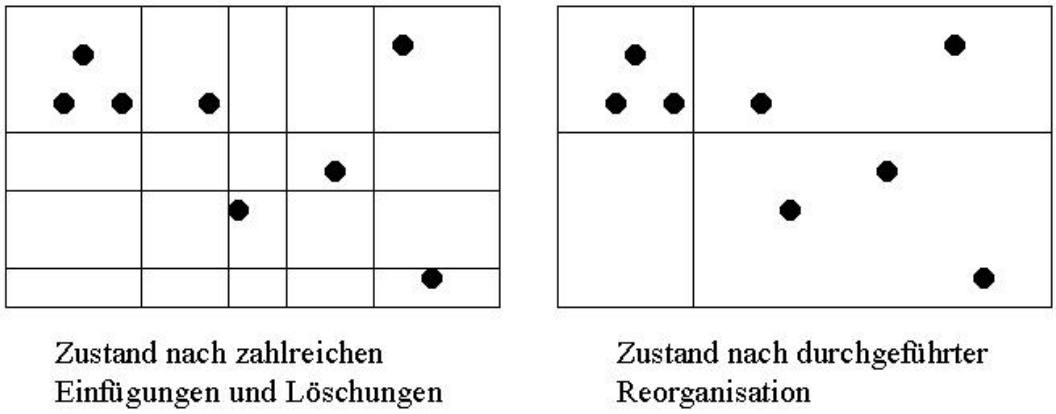


Abbildung 5: Dynamische Reorganisationsmaßnahmen

2.4.2 Hauptspeicher-basierte räumliche Indexstrukturen

Dieser Klasse gehören alle räumlichen Indexstrukturen an, bei deren Entwurf nicht auf eine Sekundärspeicherverwaltung Wert gelegt wurde. Dies bedeutet in anderen Worten, daß eine effiziente Speicherung bzw. Indexierung von räumlichen Informationen ausschließlich im Hauptspeicher erfolgt und keine Vorkehrung etwa zur effizienten Aus- und Einlagerung von Daten auf Sekundärspeicher getroffen wurde. Die Vertreter dieser Klasse wurden meist in den frühen Jahren der Erforschung dieses Gebiets entworfen, doch sie sind häufig noch als

Grundstrukturen in neueren Vorschlägen erkennbar. Aber auch in ihrer "reinen" Form haben die Vertreter dieser Klasse noch durchaus praktischen Wert, wie sich u.a. in Kapitel 5 zeigt.

1) kd-Baum

1975 erstmals vorgestellt [BEN75], gehört der kD-Baum heute zu den bekanntesten Indexstrukturen für mehrdimensionale Daten. Die hier dargestellten Eigenschaften eines kd-Baums gehen auf diesen ersten Vorschlag zurück; es existieren zusätzlich noch zahlreiche, auf diesen Vorschlag basierende modifizierte kd-Baum-Strukturen.

Das grundlegende Organisationsprinzip dieser Indexstruktur ist die rekursive Aufteilung des *k*-dimensionalen Universums durch *(k-1)*-dimensionale, alternierende Ebenen. Diese Ebenen müssen dabei isoorientiert, d.h. parallel zu der durch die entsprechenden *(k-1)* Koordinatenachsen aufgespannten Ebene sein. Jede der zur Aufteilung des Universums eingesetzten Ebenen muß weiterhin mindestens einen Datenpunkt enthalten, dieser wird bei nachfolgenden Operationen auf dem kD-Baum als Referenzpunkt verwendet. Knoten eines kD-Baums enthalten damit den angesprochenen Datenpunkt und zwei Zeiger auf mögliche Nachfolger ("links" und "rechts", siehe weiter unten). Diese sind unbesetzt (`null`), wenn der Knoten ein Blatt des Baumes ist, im Falle eines inneren Knoten ist wenigstens einer dieser Zeiger gesetzt.

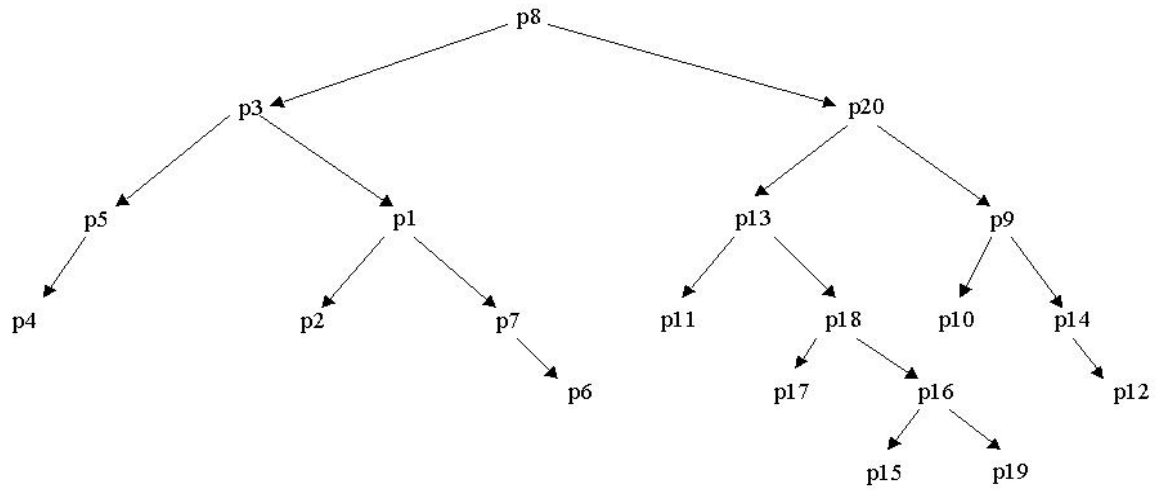
Ein Beispiel zur Verdeutlichung: Im dreidimensionalen Fall könnte die erste Aufteilung durch eine Ebene erfolgen, die von der *x*- und der *y*-Achse aufgespannt wird. Die zweite Aufteilung könnte dann durch eine die *x*- und *z*-Achse vollständig enthaltende Ebene durchgeführt werden. Die dritte Aufteilung des Universums müsste dann durch eine die *y*- und *z*-Achse enthaltende Ebene erfolgen. Als nächstes würde dann wieder die erste Aufteilung durchgeführt werden usw.. Diese strikt zyklisch alternierende Auswahl von den Raum partitionierenden Hyperbenen wird in [FRI77] durch eine mehr datenkorrelierte Aufteilung ersetzt, nämlich die aufteilende Hyperebene im Median aller Werte der Dimension aufzuspannen, deren Werte die größten Unterschiede innehatten. Moore [MOO91] schlug daraufhin vor, den Median durch das arithmetische Mittel zu ersetzen, um dadurch eine noch gleichmäßigere Aufteilung zu erzielen.

Das Halten eines Datenpunktes zu Referenzzwecken bedingt die Eigenschaft eines kD-Baums, die Daten in allen Knoten zu halten und nicht nur in den Blattknoten, wie dies bei

vielen anderen Baumstrukturen der Fall ist. Dies, zusammen mit der Unterscheidung nach nur jeweils einer Dimension in jedem Knoten, führt unter anderem aber auch dazu, daß in der Realität nahe zusammenliegende Punkte im Baum unter Umständen in weit auseinanderliegenden Knoten gespeichert werden.

Wenn nun ein neuer Datenpunkt eingefügt werden soll, dann wird zunächst der Baum auf der Suche nach einem geeigneten freien Platz vom Wurzelknoten ausgehend durchlaufen. Für jeden Baumknoten, der hierbei besucht wird, wird geprüft, ob sich der einzufügende Datenpunkt "links" oder "rechts" der den Datenpunkt dieses Knotens enthaltenden *(k-1)*-dimensionalen Ebene befindet. Im ersten Fall wird der Pfad im Baum über den ersten Nachfolgerknoten fortgesetzt, im zweiten Fall über den zweiten Nachfolgerknoten. Dieses Einfügeverhalten macht deutlich, daß der spätere Aufbau eines kd-Baums abhängig ist von der Reihenfolge, in der die in ihm gehaltenen Datenpunkte eingefügt wurden. Anders gesagt, können zwei kd-Bäume, in denen dieselbe Menge von Datenobjekten gehalten werden, einen völlig verschiedenen Aufbau haben.

Löschvorgänge gestalten sich beim k-D-Baum umständlich, denn es ist in den meisten Fällen nötig, die Datenpunkte aller Knoten, die sich im Baum unterhalb des Knotens mit dem zu löschenden Datenpunkt befinden, neu in den Baum einzufügen.
Der mögliche Aufbau eines kD-Baumes ist in untenstehender Abbildung zur Veranschaulichung dargestellt.



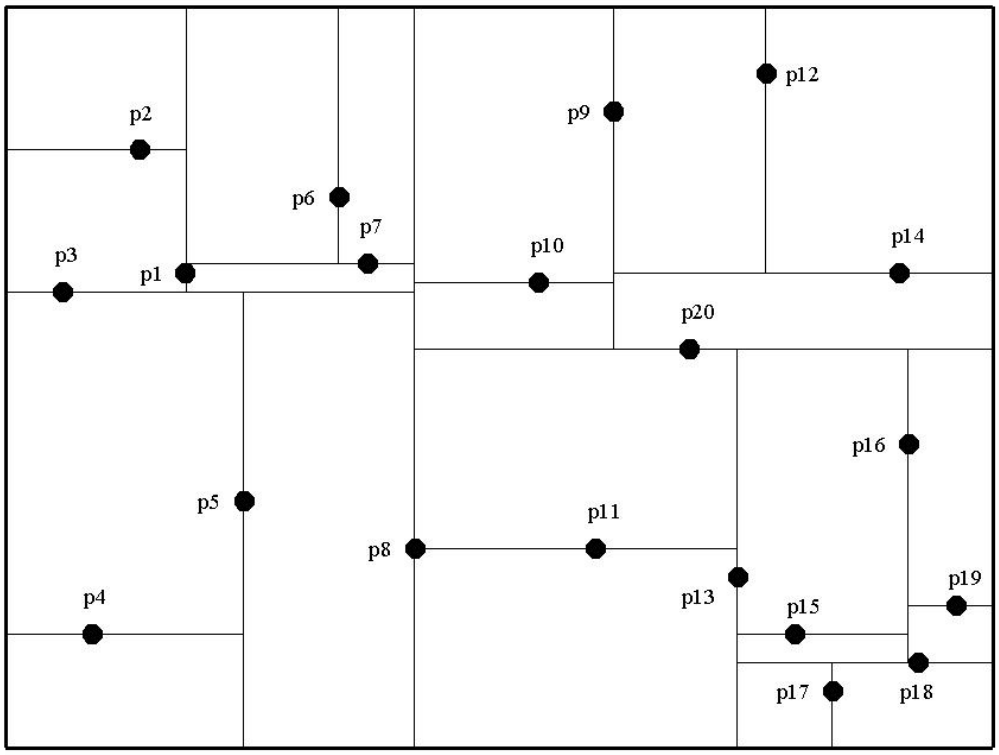


Abbildung 6: Möglicher Aufbau eines kd-Baumes für die Beispielsobjektverteilung

2) Quad-Tree

Der (Point-)Quad-Tree wurde 1974 in [BEN74] erstmals vorgestellt, eine abgeänderte Version wurde in [SAM84] präsentiert, der sogenannte Region-Quad-Tree. Unabhängig von seiner Ausprägung ist der Quad-Tree in seiner Konzeption dem kD-Baum in vielem ähnlich, doch beeinhaltet er einige grundlegende Veränderungen.

Obwohl der Quad-Tree eigentlich für die Verwaltung beliebiger n-dimensionaler Daten entworfen wurde, impliziert sein Name die Verwendung im zweidimensionalen Kontext, nämlich das rekursive Aufteilen des Universums in je vier Teile. Im folgenden soll bei der Beschreibung deshalb der zweidimensionale Fall angenommen werden, es sei aber angemerkt, daß das Prinzip des Quad-Trees auch für drei- oder mehrdimensionale Universen angewendet werden kann. Im Falle eines dreidimensionalen Quad-Trees spricht man auch von einem Oct-Tree, dieser wird im Bereich des CAD (Computer Aided Development) häufig eingesetzt.

Jeder Knoten eines Quad-Trees enthält mindestens eines Datenpunkt und vier Zeiger auf Nachfolger. Diese identifizieren jeweils ein bestimmtes Teilgebiet des von diesem Knoten abgedeckten Auschnitts des Universums. Jedes dieser Teilgebiete wird durch isoorientierte

Ebenen (im zweidimensionalen Fall also Geraden) begrenzt. Für innere Knoten hat jeder dieser Zeiger einen von `null` verschiedenen Wert, bei Blattknoten dagegen besitzt jeder den Wert `null`.

Das Einfügen eines Datenpunktes gestaltet sich nun so, daß beginnend beim Wurzelknoten für jeden besuchten Knoten zunächst geprüft wird, ob dieser ein Blattknoten ist. Trifft dies zu, so wird getestet, ob dieser noch Platz bietet, um den Punkt aufzunehmen. Ist dies der Fall, so wird der Punkt eingefügt und der Vorgang beendet. Im anderen Fall wird das von diesem Knoten überdeckte Gebiet in vier Gebiete aufgeteilt, für jedes dieser Gebiete ein neuer Blattknoten erzeugt, diese über die vier Zeiger des alten Blattknotens in den Baum eingebunden und schließlich alle Datenpunkte des alten Blattknotens zusammen mit dem einzufügenden Datenpunkt auf die vier neuen Blattknoten verteilt. Der alte Blattknoten wird nun als innerer Knoten behandelt. Wenn der aktuelle Knoten aber kein Blattknoten ist, so wird geprüft, welches der vier Teilgebiete sich eignet, um den Punkt später aufzunehmen und der Pfad durch den Baum wird durch Verfolgen des entsprechenden Zeigers fortgesetzt. Dieses Vorgehen zeigt einen wesentlichen Unterschied zum zuvor betrachteten kD-Baum auf, im Gegensatz zu diesem hält der Quad-Tree alle Datenpunkte in seinen Blättern und nicht über den ganzen Baum verteilt. Im Falle des Point-Quad-Trees wird die oben erwähnte Aufteilung durch Aufspaltung des zu teilenden Gebiets in vier möglicherweise unterschiedlich große Teilgebiete durchgeführt. Im Zentrum dieser Gebiete steht dann der einzufügende Punkt, so daß die vier entstehenden Teilgebiete von unterschiedlicher Größe sein können. Im Falle des Region-Quad-Trees führt die Aufteilung dagegen immer zu vier gleichgroßen Teilgebieten, was spätere Suchvorgänge im Baum erheblich erleichtert, aber Nachteile bezüglich der Anpassungsfähigkeit an ungleichmäßige Datenverteilungen hervorruft.

In untenstehender Abbildung soll der Aufbau eines (Region-)Quad-Trees nochmals veranschaulicht werden.

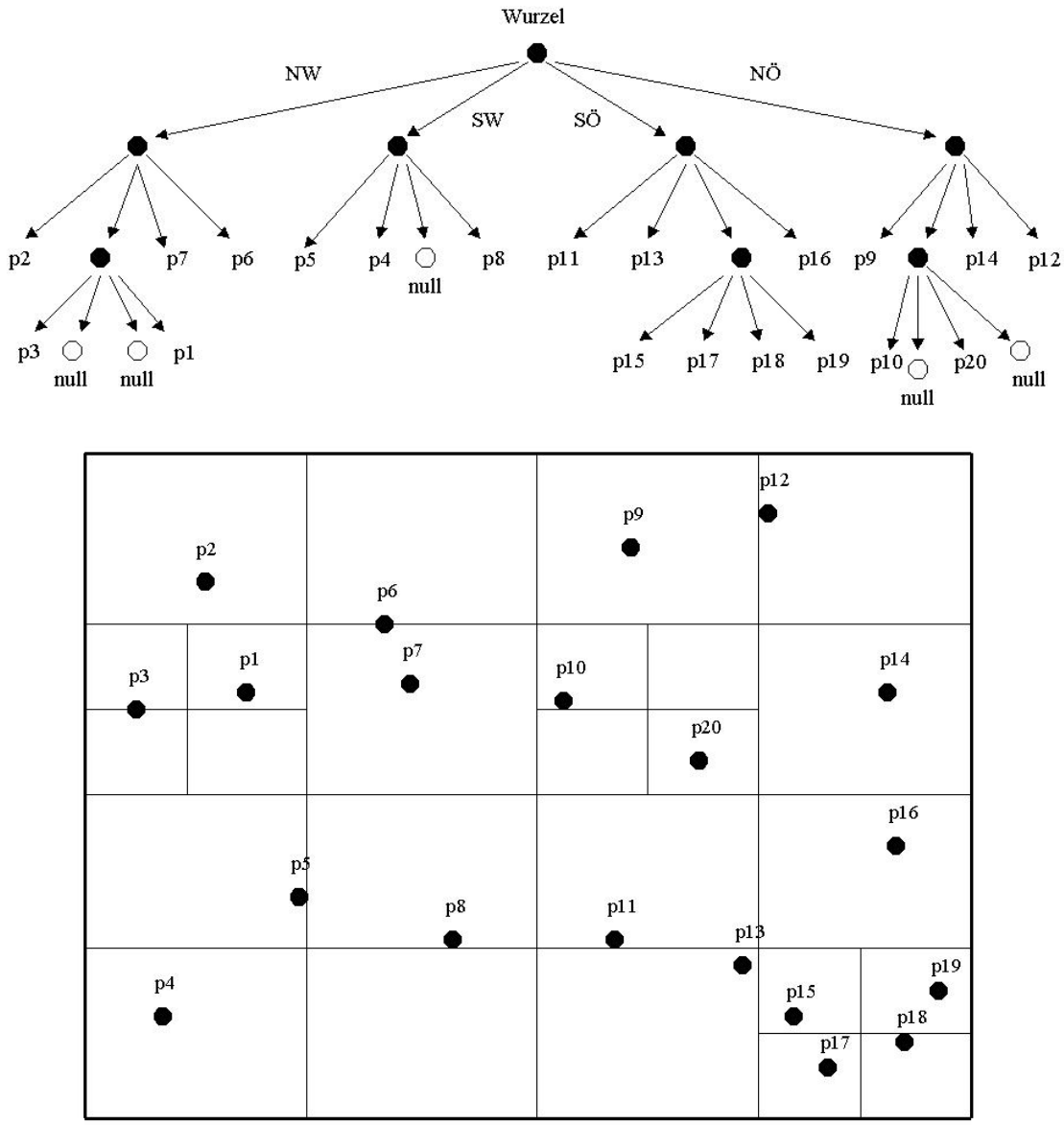


Abbildung 7: Möglicher Aufbau eines Region-Quad-Trees für die Beispielobjektverteilung

2.4.3 Point Access Methods

Angehörige dieser Klasse haben die Gemeinsamkeit, daß sie den Zugriff auf punktförmige Objekte ohne räumliche Ausdehnung unterstützen und außerdem für die effiziente Nutzung des Sekundärspeichers konzipiert wurden. Dies umfasst in den meisten Fällen die Unterstützung der Aus- und Einlagerung von Datenobjekten in Seiten auf den Sekundärspeicher, so daß über die sich im Hauptspeicher befindliche räumliche Indexstruktur möglichst effizient auf die eigentlichen Datenobjekte zugegriffen werden kann. Dies ist vor allem dann von Vorteil, wenn große Mengen an Daten verwaltet werden müssen und daher eine Auslagerung auf Sekundärspeicher vonnöten ist. Natürlich können die Vertreter dieser Klasse bei Bedarf auch ohne Verwendung der Sekundärspeichermechanismen implementiert

werden, wie z.B. in Kapitel 4 zu sehen sein wird. Die Beschreibung der nachfolgenden Indexstruktur stellt diese aber in ihrer ursprünglichen Konzeption, also mitsamt des Sekundärspeichermechanismus vor. Angehörige dieser Klasse unterstützen generell keine Datenobjekte mit räumlichen Ausdehnung wie z.B. Polygone.

1) Grid-File

Die räumliche Indexstruktur "Grid-File" wurde 1981 in [NIE84] vorgestellt und war bis heute Basis vieler ähnlicher Vorschläge, unter anderem auch dem des weiter unten vorgestellten Multilayer-Grid-Files.

Das Grundprinzip des Grid-Files ist es, dem Universum ein d-dimensionales orthogonales Raster aufzuprägen, ähnlich dem Kästchenraster in einem Rechenheft für den zweidimensionalen Fall. Da das Raster nicht notwendigerweise regulär ist, kann jedes Rasterfeld, im folgenden mit Zelle bezeichnet, eine unterschiedlich große Fläche (dreidimensional: Volumen) besetzen.

Der oben angesprochene Mechanismus zur effektiven Auslagerung und Zugriff auf Datenobjekte in bzw. aus dem Sekundärspeicher wird beim Grid-File dermaßen realisiert, daß über ein sogenanntes Rasterverzeichnis (engl.: Grid Directory) jede der Zellen mit genau einer Speicherseite assoziiert wird. Dabei gilt eine *n:1*-Beziehung, es können nämlich auch mehrere benachbarte Zellen einer Seite zugewiesen werden, jedoch nicht mehrere Seiten einer Zelle. Sollte das Rasterverzeichnis zu groß werden, um es durchgehend im Hauptspeicher zu halten, so kann es auch auf den Sekundärspeicher ausgelagert werden. Im Hauptspeicher resident ist dann nur noch das Raster selbst, das durch *d* Felder von eindimensionalen Ebenen, den Skalen (engl.: Scales), beschrieben wird.

Für Suchvorgänge wird nun zuerst in der Menge der Skalen nach dem Verweis auf die Zelle gesucht, die den Suchpunkt enthalten müßte. Dazu muß in jeder Skala nach dem Index gesucht werden, den die entsprechende Komponente des einzufügenden Punktes innehaben würde. Aus allen gefunden Indizes kann dann die richtige Zelle bestimmt werden.

Ein kleines Beispiel dazu: Eingefügt werden soll der Punkt mit den kartesischen Koordinaten *(3,5)*. Die horizontale Skala enthält bereits die Werte *1,2* und *4*, die vertikale Skala die Werte *0* und 6. Dann wird als horizontaler Index der Wert *2* zurückgeliefert, als vertikaler Index der

Wert *1*, da der x-Wert von *3* zwischen *2* und *4* liegt, also in der zweiten horizontalen Rasterzone und der y-Wert dementsprechend in der ersten und bislang einzigen vertikalen Rasterzone. Diese beiden Indizes identifizieren dann die gesuchte Zelle eindeutig.

Ist die betreffende Zelle momentan nicht im Hauptspeicher, so wird diese vom Betriebssystem in diesen geladen, dies kostet einen Sekundärspeicherzugriff. Die geladene Zelle enthält einen Zeiger auf eine Speicherseite, an der der gesuchte Datenpunkt stehen wird, sofern er existiert. Der Zugriff auf diese Speicherseite kann nochmals einen Sekundärspeicherzugriff erfordern, sofern sie sich nicht bereits im Hauptspeicher befindet. Insgesamt werden somit nur maximal zwei Sekundärspeicherzugriffe benötigt.

Das Einfügen eines Punktes gestaltet sich ähnlich: Zuerst wird wie oben die Zelle gesucht, in der sich der Datenpunkt befinden muß. Über die Zelle wird die Speicherseite gefunden, die mit dieser Zelle assoziiert ist. Ist noch Platz in dieser Zelle, so wird der Datenpunkt dort eingetragen, im anderen Fall muß zwischen zwei Fällen unterschieden werden. Im ersten Fall, wenn mehrere Zellen mit einer Seite assoziiert sind, wird geprüft, ob die Skalen eine Ebene enthalten, durch die die betreffende Speicherseite so geteilt werden kann, daß sich abhängig von der jeweiligen Implementierung wenigstens ein Datenpunkt, inklusive dem einzufügenden, in jeder der zwei neuen Teile befindet. Ist dies der Fall, so wird eine neue Seite allokiert und die Datenpunkte entsprechend auf die beiden Seiten verteilt. Ist keine der in den Skalen vorhandenen Ebenen geeignet, die Speicherseite zu teilen, oder nur eine Zelle mit der Speicherseite assoziiert (Fall zwei), so wird eine neue teilende Ebene eingeführt und eine neue Speicherseite allokiert. Alle Datenpunkte, inklusive dem einzufügenden, werden dann entsprechend auf die beiden Seiten verteilt, die neue Ebene wird in die entsprechende Skala eingetragen und alle Zellen, die von dieser Ebene geschnitten werden, werden dementsprechend ebenfalls geteilt.

Nach [NIE84] sollten auf dem *k*-dimensionalen Rasterverzeichnis folgende Operationen für effizientes Ausführen von Akualisierungen und Anfragen definiert sein:

- Direkter Zugriff auf einen Eintrag des Rasterverzeichnisses.
- Relativer Zugriff in jede der möglichen Dimensionen ausgehend von der momentanen Position („Eine Zelle in Dimension *x* höher“, „eine Zelle in Dimension *y* tiefer“).
- Mischen zweier benachbarter Zellen einer Dimension.

- Partitionieren einer Zelle in zwei Teile.

Wie bereits oben angesprochen, ist die Sekundärspeicherverwaltung der PAMs lediglich eine Option. Auf diese kann gegebenfalls auch verzichtet werden, etwa wenn aus Leistungsgründen ganz auf den Zugriff auf den Sekundärspeicher verzichtet wird und die Anzahl der zu verwaltenden Daten eine bestimmte Schwelle nicht überschreitet.

Untenstehende Abbildung verdeutlicht die Funktionsweise des Grid-Files durch Darstellung dessen Rasterverzeichnisses noch einmal.

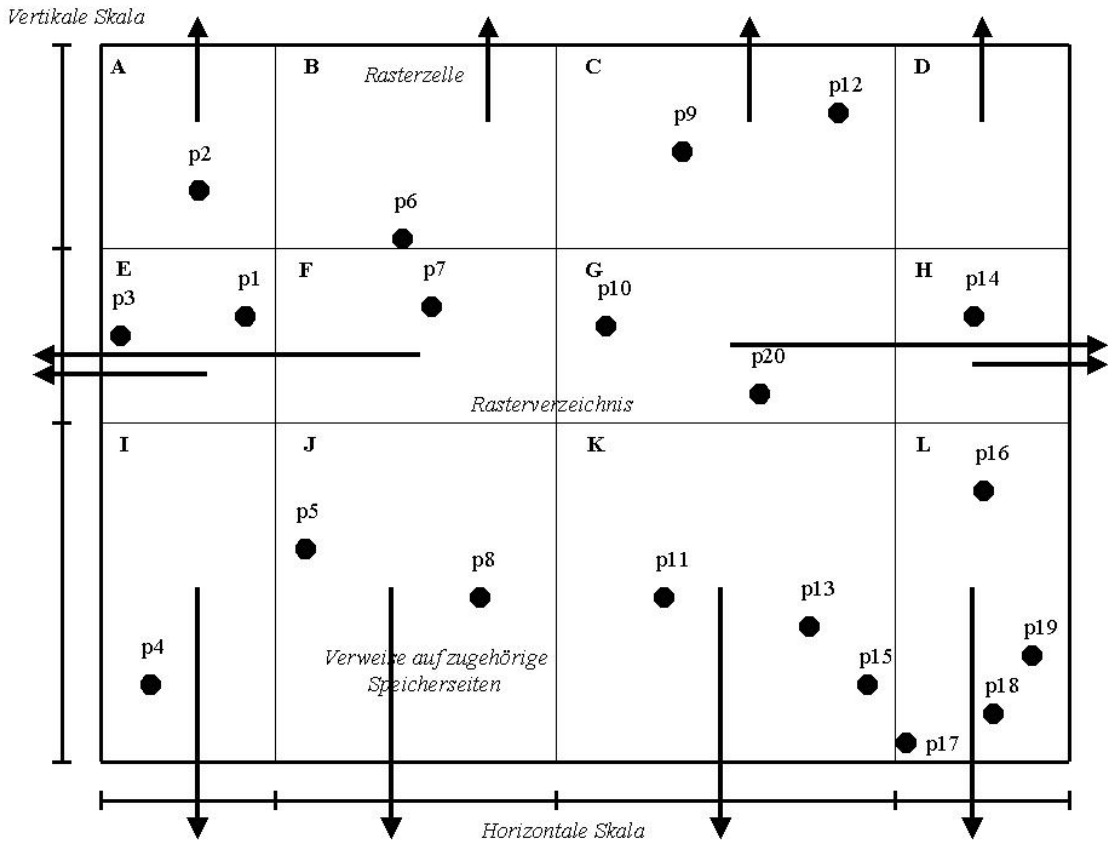


Abbildung 8: Rasterverzeichnis eines Grid-Files für die Beispielobjektverteilung

2.4.4 Spatial Access Methods

Die Mitglieder der letzten Klasse sind im allgemeinen die Ergebnisse jüngerer Forschungsarbeiten. Sie wurden für den Zweck entworfen, statt nur punktförmiger auch Datenobjekte mit räumlicher Ausdehnung zu verwalten und dadurch den effizienten Zugriff auf diese zu ermöglichen. Den möglichen großen Datenmengen moderner Anwendungen wird

durch eine effiziente Sekundärspeicherverwaltung, wie bei den PAMs, Rechnung getragen. Wie bei den PAMs auch, ist es selbstverständlich gegebenenfalls möglich und auch vorteilhaft, eine SAM ohne diese Sekundärspeicherverwaltung zu implementieren. SAMs können auch als Erweiterung der PAMs gesehen werden, deren Einsatzgebiet durch Anwendung einer der folgenden Techniken erweitert wurde:

- Transformation,
- Überlappung von Gebieten,
- Clipping oder
- multiple Ebenen

Im folgenden werden nur die Techniken "Überlappung von Gebieten" und "multiple Ebenen" kurz umrissen, für eine tiefergehende Betrachtung und eine umfassende Betrachtung einer Vielzahl von räumlichen Indexstrukturen siehe z.B. [GAE98].

- *Überlappung von Gebieten:* Zur Behandlung räumlich ausgedehnter Objekte wird die gegenseitige Überlappung von Speicherseiten erlaubt. Das bedeutet, daß ausgedehnte Objekte als ganzes in einer Speicherseite gehalten werden können und nicht auf mehrere Seiten aufgeteilt werden müssen. Nachteilig ist hier allerdings, daß zur Beantwortung einer Anfrage nun u. U. mehr Seiten geprüft werden müssen. Der nachfolgend beschriebene R-Baum wurde mit dem Gestaltungsgrundsatz „Überlappung von Gebieten“ entworfen.
- *Multiple Ebenen:* Dieser Ansatz basiert auf der Verwendung mehrerer hierarchisch angeordneter Indexschichten, wobei jede dieser das Universum auf unterschiedliche Weise aufteilt. Überlappungen innerhalb einer Schicht sind nicht erlaubt, oftmals ist die Art der Aufteilung für jede Ebene im voraus festgelegt. Die Indexstruktur „Multilayer-Grid-File“ wurde beispielsweise gemäß den Grundsätzen der multiplen Ebenen entworden.

1) R-Baum

Der R-Baum, im Jahre 1984 von Guttman vorgestellt, ist der wohl bekannteste Vertreter der Klasse der SAMs und basiert auf dem bekannten B-Baum für eindimensionale Daten [BMC72]. Es wurde immense Forschungsarbeit für die Weiterentwicklung dieser Struktur geleistet, dies äußerte sich in Varianten wie dem R*-Baum [BEC90], oder auch exotischeren

Varianten wie dem Hilbert-R-Baum [KAF94] oder dem X-Tree [BER96]. Im wesentlichen unterscheiden sich all diese Varianten aber lediglich durch ihre Knoten-Aufteilungstrategien, das Grundkonzept des R-Baums, wie nachfolgend beschrieben, hat sich dabei kaum verändert.

Der R-Baum ist eine hierarchische Baumstruktur, deren Funktionsprinzip auf der Schachtelung *d*-dimensionaler Intervalle, die auch als minimal überdeckende Rechtecke (engl.: Minimum Bounding Boxes, kurz MBB) bezeichnet werden, beruht. Eine MBB ist die kleinstmögliche, isoorientierte Umschreibung einer beliebigen geometrischen Figur. Sie wurden eingeführt, um alle möglichen Arten von Polygonen in einem R-Baum einfach verwalten zu können. Zur Veranschaulichung dient die folgende Abbildung.

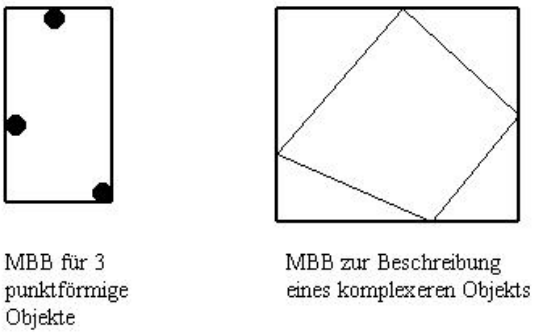


Abbildung 9: Beispiel für Minimum Bounding Boxes

Jeder Knoten eines R-Baums deckt damit ein bestimmtes *d*-dimensionales Intervall ab und wird mit einer eigenen Speicherseite assoziiert. Er enthält je einen Verweis auf mindestens zwei Nachfolgerknoten, wobei der MBB dieses Knotens die MBBs der Nachfolger vollständig enthalten. Der MBB eines Knotens umschreibt alle in diesem Knoten gehaltenen geometrischen Gebilde. Dies sind im Falle eines inneren Knotens die MBBs der Nachfolger und im Falle eines Blattknotens die Umrisse der dort gespeicherten Polygone. Die MBBs der Knoten auf einer Ebene des R-Baums können überlappen. Jeder Knoten eines R-Baums enthält zwischen *m* und *M* Einträge, außer der Wurzel, die, wenn sie selbst kein Blatt ist, mindestens zwei Einträge enthält.
Eine weitere Eigenschaft des R-Baums ist die Höhenbalanciertheit, diese beträgt zumeist $|log_m(N)|$ für *N* innere Knoten *(N > 1)*. Dies drückt sich auch in der Tatsache aus, daß alle Blätter auf einer Baumebene wachsen.

Ein Suchvorgang in einem R-Baum läuft nun so ab, daß ausgehend von der Wurzel die MBB jedes Nachfolgerknotens auf Schnitt mit dem Suchintervall überprüft wird (man bemerke den grundlegenden Unterschied zwischen SAMs und PAMs: Während PAMs nur Suchen nach Objekten ohne räumliche Ausdehnung unterstützen, bieten SAMs wie der R-Baum die Möglichkeit zusätzlich auch nach räumlich ausgedehnten Objekten zu suchen). Ist die Schnittmenge ungleich der leeren Menge, so wird rekursiv in dem entsprechenden Nachfolger gesucht. Da Überlappungen der MBBs der Knoten einer Ebene ausdrücklich zugelassen sind, kann dies zur Verfolgung mehrerer Pfade bei solchen Suchvorgängen führen. Da im R-Baum üblicherweise nur die MBB eines Datenobjekts gehalten wird und nicht das Datenobjekt selbst, müssen bei Erreichen eines Blattes während eines Suchvorganges die dort referenzierten Datenobjekte in den Hauptspeicher geladen und auf Schnitt mit dem Suchintervall getestet werden. Erst wenn auch dieser Test erfolgreich ausgefallen ist, wird das entsprechende Datenobjekt in die Ergebnismenge aufgenommen.

Zum Einfügen eines Datenobjekts muß ausgehend von der Wurzel ein geeignetes Blatt gefunden werden. In diesem wird dann die MBB des Objektes sowie eine Referenz auf dieses gehalten. Der weiter zu verfolgende Pfad wird bei jedem besuchten inneren Knoten bestimmt, indem der Nachfolgerknoten ausgewählt wird, dessen MBB die geringste Vergrößerung bei Einfügen der MBB des Datenobjektes erfahren würde (Politik der geringsten Vergrößerung, engl.: Least Enlargement Policy). Um Mehrdeutigkeiten aufzulösen, schlägt Guttman vor, bei mehreren qualifizierten Nachfolgerknoten denjenigen auszuwählen, der die kleinste MBB besitzt. Die untenstehende Abbildung verdeutlicht das soeben beschriebene Vorgehen noch einmal.

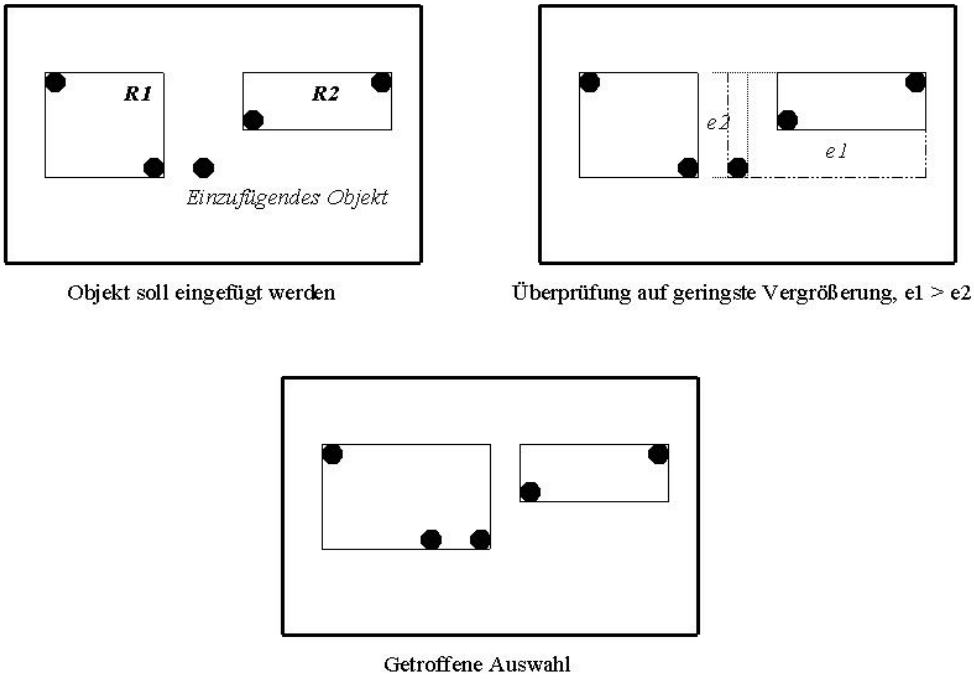
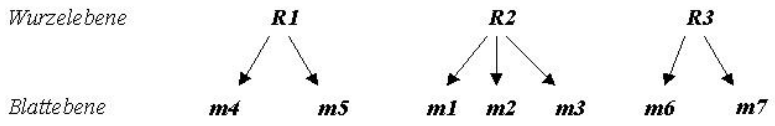


Abbildung 10: Beispiel für Vorgänge beim Einfügen eines Objektes in einen R-Baum

Bei Erreichen eines Blattes wird geprüft, ob noch Platz vorhanden ist, um das Datenobjekt aufzunehmen. Ist dies der Fall, so wird der MBB und die Referenz auf das eigentliche Objekt dort gespeichert, der MBB des Blattes angepaßt und die Änderungen bis zur Wurzel nach oben propagiert. Im anderen Fall wird ein neues Blatt erzeugt, und alle Einträge, inklusive dem neuen Eintrag, auf diese beiden Blätter verteilt, deren MBB berechnet und die Änderungen wieder aufwärts propagiert.

Das Löschen eines Elements gestaltet sich wie folgt: Wenn das Datenobjekt durch einen Suchvorgang gefunden wird, so wird es aus dem betreffenden Blatt gelöscht. Wenn eine gewisse Anzahl von Elementen nicht unterschritten wird, so werden die Änderungen lediglich aufwärts propagiert. Im anderen Fall werden Restrukturierungsmaßnahmen angestoßen, um eine gewisse Mindestbelegung zu garantieren (siehe hierzu auch Kapitel 4.5).

Zur Veranschaulichung ist abschließend zu dieser Betrachtung in untenstehender Abbildung ein R-Baum dargestellt.



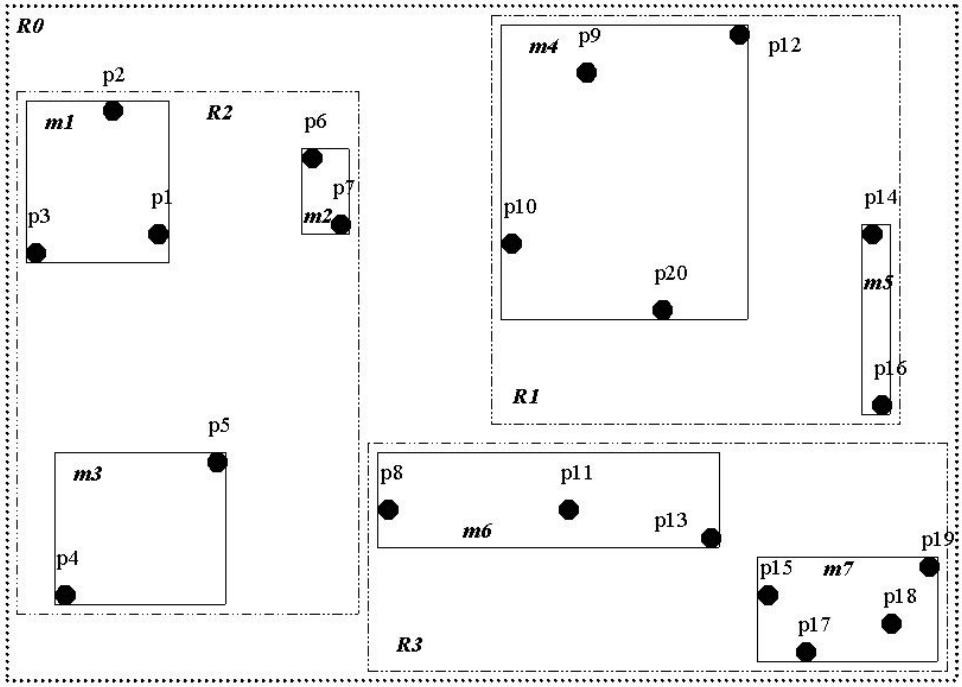


Abbildung 11: Aufbau eines R-Baums für die Beispielobjektverteilung

2) Multilayer Grid-File [SIX88]

Diese Indexstruktur basiert auf dem bereits vorgestellten Grid-File, ist aber im Gegensatz zu diesem nicht nur aus einem, sondern aus mehreren hierarchisch angeordneten Rastern aufgebaut. Jedes dieser Raster stellt dabei eine unabhängige Schicht in der gesamten Indexstruktur dar.

Einfügeoperationen werden nun üblicherweise so implementiert, daß zuerst versucht wird, ein räumliches Datenobjekt in das Raster der untersten Schicht einzufügen. Dies gilt als gelungen, wenn dieses Datenobjekt nicht von einer teilenden Ebene geschnitten wird. Ist dies aber der Fall, so wird versucht, das Objekt in der nächsthöheren Schicht einzufügen. Dies wird fortgesetzt, entweder, bis eine geeignete Schicht erreicht wird, oder bis zum Erreichen der obersten Schicht, in der Schnitte zwischen teilenden Ebenen und Datenobjekten notwendigerweise erlaubt sind. Wenn die jeweilige Implementierung des Mulitlayer-Grid-Files das nachträgliche Einfügen von teilenden Ebenen zuläßt, so muß ein in einer Schicht eingetragenes Datenobjekt unter Umständen nachträglich in die nächsthöhere Schicht aufsteigen, da durch das Einfügen die angesprochene Schnittbedingung verletzt sein könnte. Möglich ist es in diesem Zusammenhang aber auch, das nachträgliche Einfügen von teilenden

Ebenen nicht zu gestatten. Damit würde von Beginn an die Granularität der Skalen jeder Schicht festgelegt, wobei die unterste Ebene die feinste und die oberste Schicht die gröbste Granularität besitzen würde. Sicherlich ist diese Politik weniger flexibel, allerdings muß sich die Implementierung nicht mehr um das nachträgliche Aufsteigen von Datenobjekten in höhere Schichten kümmern.

Suchvorgänge nach einem Datenobjekt gestalten sich nun so, daß anhand der räumlichen Ausdehnung des Suchobjektes auf die Schicht in der dieses voraussichtlich abgelegt wurde, zurückgeschlossen wird und mit der Suche direkt im Rasterverzeichnis dieser Schicht begonnen wird. Für Suchvorgänge des Typs "Bereichsanfrage" oder "Positionsanfrage" (siehe Kapitel 2.3) wird stattdessen mit der Suche auf der untersten Schicht begonnen und diese nach oben hin fortgesetzt.

Das Durchführen von Reorganisationsmaßnahmen kann neben dem beim Grid-File üblichen Verschmelzen schlecht besetzter benachbarter Zellen so gestaltet werden, daß Objekte zusätzlich so weit wie möglich nach unten verschoben werden, um so die oberste Schicht zu entlasten. Hier sollte mit dem Verschieben von „kleinen“ Objekten begonnen werden, da so die Wahrscheinlichkeit höher ist, daß mehrere Objekte verschoben werden können. Die Durchführung solcher Maßnahmen ist allerdings sehr aufwendig, da dafür wiederum viele Lösch- und Einfügeoperationen nötig sind, die selbst wieder unter Umständen zu schlecht besetzten Zellen führen könnten.

Das Multilayer-Grid-File soll mit der umseitigen Abbildung nachfolgend noch einmal vereinfacht grafisch dargestellt werden.

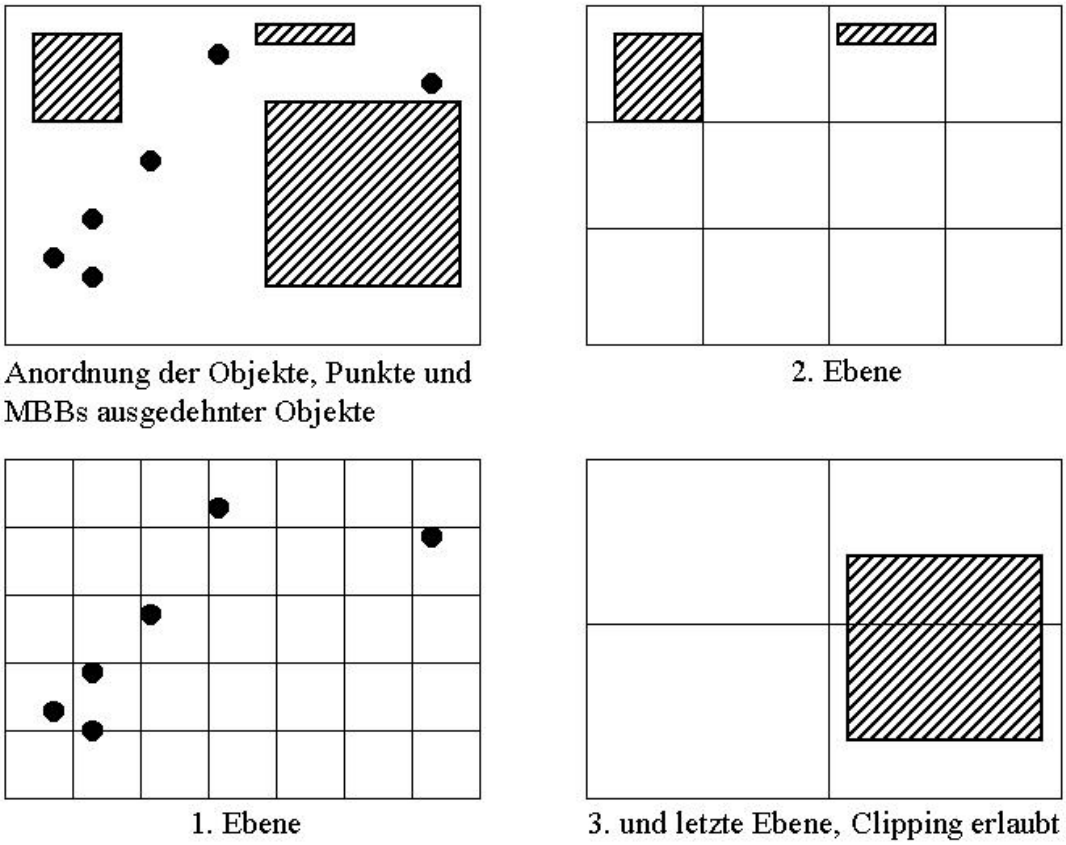


Abbildung 12: Aufbau eines Multilayer Grid-Files

3 Anforderungen des Lokationsdienstes

In diesem Kapitel soll ein Katalog von wünschenwerten Eigenschaften augestellt werden, die eine räumliche Indexstruktur besitzen sollte, um für die Speicherung der anfallenden räumlichen Daten des Lokationsdienstes des NEXUS-Projektes geeignet zu sein.

Hierzu wird zunächst einmal die Funktionalität des angesprochenen Lokationsdienstes beschrieben, um in dem darauf folgenden Unterkapitel auf allgemeine Anforderungen einzugehen die eine räumliche Indexstruktur erfüllen sollte. Diese Anforderungen werden im nächsten Unterkapitel auf ihr Zutreffen im Bezug auf die Eigenschaften des Lokationsdienstes untersucht, und entweder übernommen, oder verworfen.

3.1 Funktionalität des Lokationsdienstes

Innerhalb des NEXUS-Projektes [NEX99] an der Universität Stuttgart ist ein verteilter Lokationsdienst für die Verwaltung der Positionsinformationen mobiler Objekte zuständig. Ein Teil dieses Lokationsdienstes ist eine Komponente, die für die eigentliche Datenhaltung der angesprochenen Positionsinformationen verantwortlich ist, diese muß, um effektiv zu sein, auf einer räumlichen Indexstruktur basieren. Effektivität bei der Ausführung von Anfragen und Aktualisierungen ist für den Lokationsdienst äußerst wichtig, da für die Garantie einer hohen Positionsgenauigkeit das häufige Ausführen dieser Operationen nötig wird. Der Lokationsdienst besitzt zusammengefasst folgende, für die Wahl einer Indexstruktur wichtige Eigenschaften. Für eine ausführliche Beschreibung siehe [LOC01]

- Verwaltung der Positionsinformationen einer Menge mobiler Objekte, die nach erfolgter Anmeldung bei einem Server des Dienstes aktiv handeln (können selbst Anfragen stellen) und/oder passiv (können in Lösungsmenge der Anfragen anderer enthalten sein) zur Verfügung stehen können. Anfragen können zusätzlich von nicht mobilen, registrierten Objekten gestellt werden, um so etwa Antworten auf Fragen der Art „Wie weit ist ein Lastwagen eines Speditionsbetriebes von seinem Bestimmungsort entfernt“ geben zu können (Desktop-System der Spedition ist Klient). Die Positionsinformationen werden entweder von den mobilen Objekten selbst über mobile Positionsbestimmungsgeräte (z.B. GPS) oder von einer externen Einrichtung

(z.B. Active Badge) bestimmt und werden von der betreffenden Serverinstanz zentral in einer Datenbankkomponente verwaltet. Es kann nach Bedarf mehrere Serverinstanzen geben, die in einem hierarchischen Verhältnis zueinander stehen.

- Der Fokus der angebotenen Funktionalität des Lokationsdienstes liegt auf der Beantwortung einfacher räumlicher Anfragen, genauer neben Anfragen nach der Position eines Objektes die Beantwortung von Bereichsanfragen und Nächster-Nachbar-Anfragen (siehe 2.3) für punktförmige Objekte ohne räumliche Ausdehnung.

- Die maximal verfügbare Genauigkeit hängt ab von der Genauigkeit der positionsbestimmenden Geräte und der maximal möglichen Frequenz, mit der Aktualisierungsnachrichten für die Position zwischen mobilem Objekt und Serverinstanz gesendet werden können. Eine hohes Maß an Positionierungsgenauigkeit ist demnach unter anderem mit einem hohen Aktualisierungsaufwand verbunden. Dies äußert sich für eine Indexstruktur in zahlreichen Akualisierungen in kurzer Zeit.

- Die für die Verwaltung von Positionsinformationen zuständige, zu entwickelnde Komponente sollte entweder vollständig hauptspeicherbasiert sein, oder auf einer herkömmlichen Datenbank beruhen. Gründe für die erste Alternative bestehen in der schon angesprochenen zu erwartenden hohen Aktualisierungsrate der Positionsinformationen. Diese bewirkt, daß Positionsinformationen schnell veralten und eine mögliche Wiederherstellung dieser vom Sekundärspeicher nicht sinnvoll ist. Weiterhin lassen sich Positionsinformationen sehr kompakt darstellen, nämlich durch je einen Zahlwert pro Dimension und gegebenfalls eine Kennung für das unterliegende Koordinatensystem. Durch diese Kompaktheit ist eine Speicherung auch im kapazitätsbegrenzten Hauptspeicher für die meisten Anwendungen möglich. Zuletzt kann man sich Augen führen, daß die Preise für Hauptspeicher in den letzten Jahren beständig fielen, so daß in diesem Zusammenhang nicht mehr von einem Flaschenhals gesprochen werden kann. Vielmehr wird die Leistung eines Servers von der Häufigkeit der an ihn gerichteten Anfragen und Aktualisierungen und deren Ausführungszeit bestimmt, was einen weiteren Grund für eine hauptspeicherbasierte Lösung ergibt. Die Leistungsvorteile einer rein hauptspeicherbasierten Lösung gegenüber einer Sekundärspeicherlösung liegen ohnehin auf der Hand. Die

Formulierung der zweiten Alternative stellte sich aus Gründen der Simplizität, da bei Einsatz einer kommerziellen Datenbank entsprechend weniger Implementierungsaufwand nötig werden würde. Weiterhin könnte sich so bei Auftreten von Fehlern beispielsweise der vom Hersteller der verwendeten Datenbank angebotene Support ausnutzen lassen.

3.2 Allgemeine Anforderungen

Dieses Unterkapitel hat seinen Fokus in der Darstellung allgemeiner Anforderungen für eine räumliche Indexstruktur, die nicht unbedingt als anwendungsspezifisch zu bewerten sind. Der hier vorgestellte Anforderungskatalog sollte demnach von allen räumlichen Indexstrukturen so weitgehend wie möglich erfüllt werden. Manche dieser Forderungen widersprechen sich gegenseitig, deshalb ist es unmöglich, eine Indexstruktur zu entwerfen, die allen Anforderungen genügt. Es sollte daher immer im Einzelfall entschieden werden, welche der widersprüchlichen Anforderungen wichtiger ist und bei der Wahl einer Indexstruktur diese Entscheidung mitberücksichtigt werden. Im nächsten Unterkapitel werden die nachfolgend genannten allgemeinen Anforderungen dann unter den für den Lokationsdienst nach dem derzeitigen Stand wichtigen Gesichtspunkten diskutiert und entweder übernommen, oder verworfen. Hier sei aber darauf hingewiesen, daß sich der Lokationsdienst derzeit auf einem prototypischen Entwicklungsstand befindet, und demnach noch Änderungen an diesem vorgenommen werden könnten. Dies würde unter Umständen eine spätere Neubewertung des nachfolgend aufgeführten Anforderungskatalogs nötig machen.

Wichtige allgemeine Anforderungen an eine Indexstruktur zur Haltung räumlicher Daten sind nach einer in [GAE98] präsentierten Zusammenstellung aus verschiedenen Quellen unter anderem:

AF1: Stabilität bei dynamischem Zugriffsverhalten
Eine Indexstruktur sollte ihre Struktur möglichst dauerhaft bewahren können, auch bei dynamischem Wechsel der Operationen, die auf ihr ausgeführt werden. Ein dynamisches Zugriffsverhalten einer Anwendung äußert sich beispielsweise in einem gut durchmischten Operationsspektrum, dessen Zusammensetzung sich häufig ändert. Es ist in einem dynamisch

geprägten Umfeld deshalb selten, daß die Indexstruktur viele gleichartige Operationen eines Typs stapelartig hintereinander bearbeiten muß.
Durch häufige und unregelmäßige Einfüge- bzw. Löschvorgänge auf der Indexstruktur sollte diese also nicht entarten können, d.h. strukturell eine derart ungünstige Form annehmen, daß nachfolgende Operationen auf dieser Struktur nur mit schlechter Leistung ausgeführt werden können, bzw. die Speicherung dieser Indexstruktur unverhältnismäßig viel Speicherplatz in Anspruch nimmt.

AF2: Sekundärspeicherverwaltung
Eine Indexstruktur sollte in der Lage sein, die effiziente Aus- und Einlagerung von Daten vom Hauptspeicher auf Sekundärspeicher und umgekehrt durchführen zu können. Dies ist immer dann von Vorteil, wenn so große Datenmengen verwaltet werden müssen, daß der zur Verfügung stehende Hauptspeicher allein nicht mehr ausreicht.

AF3: Umfassende, gleichmäßige Unterstützung von räumlichen Operationen
Eine Indexstruktur sollte sich nicht nur für die effiziente Ausführung eines oder einiger weniger Typen von Operationen eignen, sondern möglichst eine breite Auswahl räumlicher Operationen unterstützen. Dies beeinhaltet ebenso die Forderung, daß eine Indexstruktur nicht auf einen Typ von Operation "spezialisiert" sein sollte und dafür die Leistungsfähigkeit anderer Operationen vernachlässigt wird.

AF4: Effizienz auch bei ungünstigen Datenverteilungen
Die Leistungsfähigkeit, die eine mehrdimensionale Indexstruktur bei gleichmäßig verteiltem Datenaufkommen innehat, sollte durch eventuell auftretende ungleichmäßige Verteilungen der Daten nicht übermäßig leiden. Dies bedeutet nicht, daß keinerlei Leistungsverluste auftreten dürfen, sondern daß in diesem Fall eine objektiv betrachtet angemessene Leistungsfähigkeit angeboten wird, mit der ein Weiterarbeiten in jedem Fall möglich ist. Diese Anforderung hängt mit AF1 zusammen, keinesfalls sollte durch ungünstige Datenverteilungen ein Entarten der gesamten Struktur möglich sein, was zu schwerwiegenden Leistungseinbußen führen kann.

AF5: Unkompliziertheit
Je mehr Sonderfälle eine Indexstruktur berücksichtigen muß, desto größer ist deren Fehleranfälligkeit. In jedem Fall sollte eine unkomplizierte Indexstruktur einer Indexstruktur

mit vielen Fallunterscheidungen und einzelnen Behandlungsmethoden für eventuell auftretende Spezialfälle vorgezogen werden. Im optimalen Fall existieren keine Ausnahmeregelungen, alle Daten werden durch die gleichen Methoden in gleicher Weise behandelt.

AF6: Skalierbarkeit

Die Leistungsfähigkeit einer Indexstruktur sollte nicht zu stark von deren Größe oder Auslastung abhängig sein. Anders ausgedrückt, sollte die Ausführungszeit von Anfragen und Aktualisierungen nicht wesentlich stärker wachsen, als die Anzahl der in einer Indexstruktur gespeicherten Objekte.

AF7: Effizienz der angebotenen Operationen

Eine Indexstruktur sollte die von ihr angebotenen Operationen effizient durchführen können. Effizienz bedeutet in diesem Zusammenhang die Ausführung einer bestimmten Operation in einer möglichst kurzen Zeit. Wünschenswert ist hier auch die Existenz einer oberen (logarithmischen) Schranke der Ausführungszeit für alle möglichen Datenverteilungen.

AF8: Günstige Speicherausnutzung

Eine Indexstruktur sollte allgemein gesprochen möglichst wenig Speicher zur Verwaltung der von ihr gespeicherten Daten verbrauchen. Dies bedeutet, daß ein vernünftiges Verhältnis zwischen der Größe der von einer Indexstruktur gehaltenen Daten und der Größe der Indexstruktur selbst existieren sollte. Indexstrukturen, die diese Forderung nicht erfüllen, sind als reine Hauptspeicherstrukturen bei niedrig begrenzter Hauptspeichergröße kaum verwendbar.

AF9: Effiziente Synchronisierung konkurrierender Zugriffe

Eine Indexstruktur sollte eine effiziente, sichere Synchronisierung der Zugriffe mehrerer Prozesse anbieten. Sollten keine besonderen effizienten Synchronisationsmethoden für eine Indexstruktur bekannt sein, so sollte zumindest der sichere Ausschluß konkurrierender wechselseitiger Zugriffe gewährleistet sein, um so Inkonsistenzen in der Datenhaltung auszuschließen.

In diesem Unterkapitel wurde eine Aufstellung von grundlegenden Anforderungen an eine räumliche Indexstruktur gegeben, deren Einhaltung unabhängig von der jeweiligen Umgebung, in der diese Struktur zum Einsatz kommt, wünschenwert wären.

Im folgenden Unterkapitel werden die gefundenen Anforderungen auf ihre Gültigkeit im Rahmen des Einsatzes im Lokationsdienst der NEXUS-Plattform analysiert und bei positivem Ergebnis in die Menge der speziellen Anforderungen übernommen, auf deren Einhaltung die zur Untersuchung ausgewählten Indexstrukturen dann später überprüft werden. Es wurde versucht, die Untersuchung auf möglichst allgemeinen Basis durchzuführen, d.h. eine Anforderung wurde nur dann übernommen, bzw. verworfen, wenn sie für den Lokationsdienst in einem allgemeinen Kontext zutraf, bzw dies nicht tat. Wenn der Lokationsdienst mit spezielleren Randbedingungen einsetzt werden sollte, wie z.B. einer erwarteten stark ungleichmäßigen Verteilung der registrierten mobilen Objekte im zur Verfügung stehenden Raum, können sich gegebenfalls Änderungen am Anforderungskatalog ergeben. Bei der Betrachtung der einzelnen allgemeinen Forderungen wird dies jeweils im Rahmen der Begründung für Übernahme bzw. Verwurf angegeben.

3.3 Lokationsdienstspezifische Anforderungen

Zu AF1 (*Stabilität bei dynamischem Zugriffsverhalten)*:
Das häufige dynamische Ausführen von Aktualisierungsoperationen auf einer Indexstruktur, also im ungünstigsten Fall das Löschen und Neueinfügen eines Objektes, ist zusammen mit einem langen Einsatzzeitraum und einer ungünstigen Objektverteilung der typische Grund für das Entarten einer Indexstruktur. Da beim Einsatz des Lokationsdienstes die Positionen der bei ihm registrierten räumlichen Objekte dynamisch erzeugt werden, d.h. nicht im voraus bekannt sind, ist aber gerade diese Bedingung erfüllt. Die allgemeine Anforderung AF1 wird dementsprechend in die Menge der spezifischen Anforderungen des Lokationsdienstes übernommen (SAF1).

Zu AF2 (*Sekundärspeicherverwaltung*):
Die Existenz einer effizienten Sekundärspeicherverwaltung ist nötig, wenn das zu erwartende Datenaufkommen eine gewisse Schwelle übersteigt, bzw. die Wiederherstellbarkeit der Daten nach einem kritischen Fehler, z.B. dem Absturz des Servers, gefordert wird (engl.: Crash

Recovery). Durch Hinzufügen von Hauptspeichermodulen kann diese Schwelle nach oben hin verschoben werden. Wegen den weiter oben schon dargestellten Gründen ist eine derartige Sekundärspeicherverwaltung im Rahmen des Lokationsdienstes aber nicht erforderlich. Dadurch bevorzugt wurden dementsprechend die hauptspeicherbasierten Indexstrukturen (siehe auch 2.4.2). Auch Indexstrukturen, die zur Klasse der PAMs oder der SAMs gehören, können, wie bereits gesagt, ohne Sekundärspeicherverwaltung implementiert werden, dies wurde dann auch auf diese Weise durchgeführt. Zum gegenwärtigen Zeitpunkt wird die allgemeine Anforderung AF2 demnach nicht in den Katalog der spezifischen Anforderungen übernommen.

Zu AF3 *(Umfassende, gleichmäßige Unterstützung von räumlichen Operationen)*:
Durch die auf in 3.1 hingewiesene, zu erwartende hohe Anzahl an Aktualisierungen pro Zeiteinheit ist das effiziente Einfügen, Löschen, sowie Aktualisieren der Positionsinformationen im Vergleich zu anderen Operationen mit räumlichen Daten besonders wichtig. Indexstrukturen, deren Leistungsfähigkeit für diese Operationen beschränkt ist, könnten bei der Bearbeitung vieler solcher Operationen zunehmend an Leistung einbüßen. Dieses Argument wird noch verständlicher, wenn man sich vor Augen führt, daß die oben genannten Operationen allesamt „schreibende“ Operationen darstellen, bei deren paralleler Ausführung durch mehrere Klienten zur Vermeidung von Inkonsistenzen Sperren in der Indexstruktur gesetzt werden müssen. Diese gesetzten Sperren verhindern bis zu ihrer Freigabe dann auch das Abarbeiten anderer nur „lesender“ Zugriffe, und verringern den Operationsdurchsatz entscheidend. Diese Anforderung konkurriert mit der oben unter AF3 angegebenen Forderung der Gleichbehandlung aller Operationen. Dementsprechend wird AF3 nicht übernommen, und dies führt zur mit SAF2 bezeichneten spezifischen Anforderung:

SAF2: Eine im Rahmen des Lokationsdiensts verwendete räumliche Indexstruktur sollte die Operationen Einfügen, Löschen und Aktualisieren der Daten eines registrierten mobilen Objekts besonders effizient ausführen können.

Zu AF4 *(Effizienz auch bei ungünstigen Datenverteilungen)*:
Die Übernahme dieser Anforderung in den Katalog der spezifischen Anforderungen hängt ab vom erwarteten Verhalten der registrierten mobilen Objekte. Im Falle einer fortlaufenden

ungleichmäßigen Verteilung der Datenobjekte ist sie relevant, im anderen möglichen Fall, einer gleichmäßigen Verteilung, eher nicht.
Über das jeweilige Bewegungsverhalten der mobilen Objekte und damit die Relevanz dieser allgemeinen Forderung muß im Einzelfall entschieden werden. Legt man ein zufälliges Verhalten der registrierten mobilen Objekte zugrunde, so ist eine gleichmäßige Verteilung anzunehmen, da jeweils die gleiche Wahrscheinlichkeit besteht, daß sich ein Objekt in eine bestimmte der möglichen Richtungen bewegt, oder stehenbleibt. Da durch dieses Verhalten keine Richtung bevorzugt wird, ist anzunehmen, daß sich keine „Häufungen“ mobiler Objekte in irgendeinem Teilgebiet ergeben können. Da in der Realität Mobilität aber eher intentionell und umgebungsabhängig bestimmt wird, als zufällig, ist so betrachtet keine vollständige Gleichverteilung zu erwarten (der Lenker eines Fahrzeugs auf einer Autobahn möchte von Punkt *A* nach Punkt *B* gelangen und muß dazu dem Verlauf der Autobahn folgen). AF4 wird damit in den Katalog übernommen, und wird im folgenden unter SAF3 betrachtet werden.

Zu AF5 *(Unkompliziertheit)*:
Diese allgemeine Anforderung wird in den Katalog als spezifische Forderung SAF4 übernommen. Die Forderung an eine Indexstruktur nach einer möglichst einfachen Organisation ohne Sonderfälle ist auch für die Verwendung dieser im Rahmen des Lokationsdienstes jederzeit von Bedeutung.

Zu AF6 *(Skalierbarkeit)*:
Diese Anforderung wurde verworfen. Durch die vom Lokationsdienst angebotene Möglichkeit, die Gesamtfunktionalität auf mehrere, hierarchisch angeordnete Server mit jeweils einer eigenen Indexstruktur zu verteilen, können bei steigenden oder bekannt großen Datenmengen mehrere Server zur Verwaltung dieser großen Datenmengen eingesetzt werden. Damit stellt sich die Forderung nach Skalierbarkeit nicht in dem Maße, wie an eine Indexstruktur, die alle anfallenden Daten allein verwalten muß.

Zu AF7 *(Effizienz der angebotenen Operationen)*:
Diese allgemeine Forderung geht als spezifische Anforderung SAF5 in den spezifischen Anforderungskatalog ein. Unabhängig von der Anwendungsumgebung ist eine effiziente Ausführung der angebotenen Operationen einer Indexstruktur immer wünschenswert. Die Existenz einer oberen (logarithmischen) Schranke für die Ausführungszeit der einzelnen Operationen ist ebenfalls von großem Vorteil, denn somit lassen sich auch eventuell

zeitkritische Vorgänge mit garantiertem (logarithmischem) Zeitaufwand ausführen. In vielen Indexstrukturen kann allerdings keine solche obere Schranke angegeben werden. Da das Anfragespektrum des Lokationsdienstes bislang nur aus einfachen Anfragetypen besteht, soll neben der effizienten Ausführung von Positionsanfragen besonders auf die Effizienz der möglichen Implementierungen der beiden Anfragentypen „Bereichsanfrage“ und „Nächster-Nachbar-Anfrage“ geachtet werden.

Zu AF8 *(Günstige Speicherausnutzung)*:
Da die ausschließlich hauptspeicherbasierte Datenhaltung in der entsprechenden Komponente des Lokationsdienstes eine der möglichen Varianten zur Gestaltung dieser Komponente war, wird diese Anforderung unter SAF6 in den Katalog übernommen. Dies deckt sich auch mit dem Verwurf von AF2.

Zu AF9 *(Effiziente Synchronisierung konkurrierender Zugriffe)*:
Da der Lokationsdienst in einem Client/Server-Umfeld zum Einsatz kommt, bzw. in seiner Funktionalität auf diesem Modell beruht, sollte eine geeignete Indexstruktur effiziente Sperrmechanismen unterstützen. Dabei bedeutet „effizient“ nicht unbedingt die Existenz komplexer Sperrmechanismen, sondern ein im Verhältnis zum Aufwand des Sperrmechanismus stehendes verbessertes Zeitverhalten bei parallelen Zugriffen von mehreren Klienten einer Serverinstanz des Lokationsdienstes. Wichtig ist hier auch die Beachtung von SAF4 (Einfachheit der Indexstruktur), d.h. das Überfrachten einer an sich einfachen Indexstruktur mit einem komplexen Sperrmechanismus kann durch den hinzukommenden größeren Bearbeitungsaufwand unter Umständen zu schlechteren Ergebnissen führen. In jedem Fall wird die allgemeine Anforderung AF9 als SAF7 in den spezifischen Katalog übernommen.

Dies führt zusammengefaßt zu folgenden Forderungen, die sich an eine für den Lokationsdienst geeignete räumliche Indexstruktur stellen:

- *SAF1: Stabilität bei dynamischem Zugriffsverhalten*
- *SAF2: Effiziente Ausführung „schreibender“ Zugriffe*
- *SAF3: Effizienz bei ungleichmäßigen Datenverteilungen*

- *SAF4: Einfachheit*
- *SAF5: Effiziente Ausübung der Operationen „Bereichsanfrage“ und „Nächster-Nachbar-Anfrage“*
- *SAF6: Günstige Speicherausnutzung*
- *SAF7: Unterstützung von Sperrmechanismen*

Im folgenden Kapitel sollen die in Kapitel 2.4 betrachteten Indexstrukturen auf ihre Verträglichkeit mit den in diesem Kapitel besprochenen Anforderungen bewertet werden, um damit zu einer Empfehlung für die Wahl einer räumlichen Indexstruktur als Datenhaltungskomponente des Lokationsdienstes zu gelangen. Damit diese Untersuchung nicht nur von theoretischer Art war, wurden die Indexstrukturen in JAVA implementiert. Mit den implementierten Strukturen wurde dann eine Reihe von Experimenten durchgeführt, um eine möglichst breite Wissensbasis über deren jeweilige Eigenschaften zu erhalten. Die erzielten Ergebnisse werden anschließend in Kapitel 5 dargestellt.

4 Betrachtung räumlicher Indexstrukturen

Die im letzten Kapitel erarbeiteten Anforderungen an die Datenhaltungskomponente des Lokationsdienstes sollen in diesem Kapitel für die Datenstrukturen „kd-Baum“, „Quad-Tree“, „Grid-File“ und „R-Baum“ auf den Erfüllungsgrad hin untersucht werden.

4.1 Einleitung / Grundlagen

Die Auswahl der nachfolgend analysierten räumlichen Indexstrukturen wurde von dem Vorsatz geleitet, einen Vertreter jeder der verschiedenen Strukturenklassen aus Kapitel 2.4 auf Eignung zu untersuchen.

Da in der Aufgabenstellung dieser Arbeit die Entwicklung einer rein hauptspeicherbasierten Komponente zur Verwaltung der anfallenden räumlichen Daten des Lokationsdienstes vorgeschlagen wurde, sind aus der hierfür am geeignetsten erscheinenden Klasse der „hauptspeicherbasierten räumlichen Indexstrukturen“ zwei Vertreter ausgewählt worden, dabei handelt es sich um den „kd-Baum“ und den „Quad-Tree“. Aus den beiden anderen Klassen, der PAMs und SAMs, wurde je ein Vertreter, nämlich im ersten Fall das „Grid-File“ und im zweiten Fall der „R-Baum“ gewählt. Diese Indexstrukturen werden allesamt häufig in der Praxis verwendet. Ein bekannter Einsatzbereich des Quad-Trees ist z.B. das Umfeld des CAD. Der kd-Baum wird oftmals in Anwendungsbereichen eingesetzt, deren Dimensionalität höher als 3 ist, beispielsweise in der Bildverarbeitung. Typische Einsatzbereiche des Grid-Files und des R-Baums sind durch deren Eignung für die Sekundärspeicher-Datenhaltung das Umfeld der Datenbanktechnologien.

Die getroffene Auswahl richtete sich nach folgenden Kriterien:

- *Auswahl der bekanntesten Vertreter einer Klasse:* Es wurden vornehmlich solche Indexstrukturen einer Klasse implementiert, die schon lange bekannt sind und dementsprechend gründlich erforscht wurden. Es existieren zahlreiche Vertreter jeder Klasse, von denen nicht alle bereits in der Praxis erprobt wurden. Die Auswahl einer gut erforschten Indexstruktur bietet demnach den Vorteil einer breit gefächerten Wissenbasis für die Untersuchung.

- *Auswahl der Kernstruktur einer Familie:* Da die Existenz vieler Indexstrukturen auf die Modifikation einer gemeinsamen Kernstruktur zurückzuführen ist, wurde beschlossen, wenn möglich diese Kernstruktur auszuwählen, um damit die grundlegende Eignung einer Familie von Indexstukturen zu untersuchen. Dies geschah aus der Überlegung heraus, daß eine für die Verwendung mit dem Lokationsdienst ungeeignete Kernstruktur zu ähnlichen Ergebnissen auch für die anderen Mitglieder deren Familie führt.
- *Abdeckung von möglichst vielen Entwurfsansätzen:* Die Mitglieder einer der oben angesprochenen Familien von Indexstrukturen gehören nicht notwendigerweise alle derselben Klasse von Indexstrukturen an. So wäre es möglich gewesen, alle Klassen durch die Auswahl von Mitgliedern einer Familie abzudecken. Dies wurde vermieden, indem sich im Zweifel für die Implementierung eines Mitglieds einer noch unberücksichtigten Familie von Indexstrukturen entschieden wurde.

Für die Betrachtung der einzelnen Datenstrukturen im Hinblick auf Anforderung *SAF6* soll einleitend noch eine genauere Definition dieser gegeben werden:

Def.: Mittlerer Speicheraufwand O_{avg} pro Element

- O_{tot} : Speicherbedarf für die Teile der Indexstruktur, die nicht direkt durch die gespeicherten Daten belegt werden
- n : Anzahl der Elemente

$$O_{avg} = \frac{O_{tot}}{n}$$

In diesem Zusammenhang von Wichtigkeit ist auch der Grad der Ausnutzung des allokierten Speicherplatzes in einer räumlichen Indexstruktur. Diese wurde definiert als:

Def: Mittlere Hauptspeicherausnutzung U_{avg}

- O_u : Insgesamt „sinnvoll“ belegter Speicher
- O_{tot} : Insgesamt allokierter Speicher (siehe oben)

$$U_{avg} = \frac{O_u}{O_{tot}}$$

Manche der betrachteten räumlichen Indexstrukturen unterscheiden ihre Knoten in Blätter und Indizes. Blätter befinden sich immer am „unteren“ Ende eines Baumes, sie haben keine Nachfolger und in ihnen werden entweder alle, oder der größte Teil der im Baum gespeicherten Elemente gehalten. Indizes dagegen befinden sich überall im Baum „über“ den Blättern. Ihre Aufgabe ist es, dafür zu sorgen, daß auf in Blättern gehaltene räumliche Daten schnell zugegriffen werden kann.

Für eine ausschließlich hauptspeicherbasierte baumartige Indexstruktur, die zwischen Indizes und Blättern unterscheidet, ist es wichtig, eine möglichst geringe Anzahl von Indexknoten im Bezug auf die Anzahl aller Knoten zu besitzen. Dies hat seinen Grund darin, daß in einem Index nach obiger Definition deutlich weniger, bzw. gar keine Elemente gehalten werden, als in einem Blatt. Bei steigender Indexzahl gegenüber der Anzahl der Blätter führt dies zu einem wachsenden Speicherverbrauch für die Indexierung der Objekte. Baumartige Indexstrukturen mit einem hohen Anteil von Indizes an der Knotenmenge haben demnach eine größere Wahrscheinlichkeit, eine schlechtere mittlere Hauptspeicherausnutzung U_{avg} und einen schlecheren Wert für den mittleren Speicheraufwand pro Element O_{avg} (siehe oben) zu besitzen.

Um für eine baumartige Indexstruktur das Verhältnis von Indizes und Blättern einschätzen zu können, wurde folgendes definiert:

Def: Verhältnis Indizes zu Blätter (Index/Blatt-Verhältnis, engl.: Index-to-Leaf-Ratio) R_{il}

- Gesamtanzahl von Indizes in einer baumartigen Indexstruktur: N_i
- Gesamtanzahl von Blättern in einer baumartigen Indexstruktur: N_l

$$U_{avg} = \frac{O_u}{O_{tot}}$$

In diesem Zusammenhang ist auch von Interesse, den mittleren Besetzungsgrad für die Blätter eines Baumes festzuhalten. Hierzu wurde folgendes definiert:

Def: Mittlerer Besetzungsgrad der Blätter U_l

- Anzahl der in den Blätter momentan gespeicherten Elemente: M_l
- Maximale Anzahl der in den Blättern speicherbaren Elemente: M_{lmax}

$$U_{avg} = \frac{O_u}{O_{tot}}$$

Da die Synchronisation von parallelen, konkurrierenden Zugriffen und das Verhalten einer Indexstruktur in diesem Zusammenhang ein wichtiges Thema in dieser Arbeit war, soll nun noch ein bekanntes, für alle Indexstrukturen implementiertes Synchronisationsverfahren beschrieben werden.

Die einfachste Möglichkeit, den wechselseitigen Ausschluß von parallelen, konkurrierenden Zugriffen auf eine gemeinsame Resource (die Indexstruktur) auszuschließen, ist es, den wechselseitigen Auschluß dieser Zugriffe durchzusetzen. Dies kann in Java einfach mittels des `synchronized`-Schlüsselwortes implementiert werden, das bei jeder Methode die in irgendeiner Form auf die Indexstruktur zugreift, unabhängig, ob dies rein lesend oder auch schreibend stattfindet, in der Signatur eingefügt wird. Als Folge davon wird bei Ausführen einer solchen Methode die betreffende, die Indexstruktur implementierende Klasse gesperrt. Diese Sperre wird erst freigegeben, wenn die betreffende Methode abgearbeitet wurde und kann dann durch eine andere Methode abermals gesetzt werden.

Der Vorteil dieser einfachen Art der Synchronisierung liegt auf der Hand: Für einen Entwickler ist es nahezu ohne Vorwissen über Synchronisierungsbelange möglich, einen sicheren Auschluß von konkurrierenden Zugriffen zu ermöglichen. Nachteilig ist hierbei allerdings, daß auch parallele Zugriffe rein lesender Art ausgeschlossen werden, die problemlos auch gleichläufig durchgeführt werden könnten. Dies kann bei zahlreichen lesenden Zugriffen zu einem Nachlassen der Leistungsfähigkeit der Indexstruktur führen und damit zu einem Nachlassen der Leistungsfähigkeit des gesamten umgebenden GIS.

Um dies zu vermeiden, wurde, wie oben bereits angesprochen, das Verfahren „Write once/ Read multiple“ für einige Indexstrukturen implementiert, für die keine besseren Verfahren bekannt sind. Dieses Verfahren basiert auf einer Unterteilung der möglichen Zugriffe in rein lesende und schreibende Zugriffe. Während rein lesende Zugriffe, wie oben schon angedeutet,

problemlos parallel durchgeführt werden können, muß bei Ausführen eines schreibenden Zugriffs die betreffende Indexstruktur zur Vermeidung von Inkonsistenzen gesperrt werden. Dies läßt sich einfach so durchsetzen:

- Erzeuge einen Zähler für die schreibenden Zugriffe, dieser hat immer entweder den Wert 0 oder den Wert 1. Das Setzen der Schreibsperre impliziert eine Zuweisung an diesen Zähler des Wertes 1, das Lösen eine Zuweisung des Wertes 0.
- Erzeuge einen Zähler für die rein lesenden Zugriffe, dieser hat immer einen Wert größer 0. Das Setzen einer Lesesperre impliziert eine Inkrementierung dieses Zählers, das Lösen eine Dekrementierung.
- Erzeuge ihrerseits synchronisierte Methoden (z.B. durch das `synchronized`-Schlüsselwort in JAVA), die jeweils für das Setzen und Lösen einer der beiden Sperren verantwortlich sind. Dabei muß gelten, daß eine schreibende Sperre von einem Prozeß nur gesetzt werden kann, wenn sowohl keine lesende Sperre vorher gesetzt und noch nicht freigegeben wurde (Lesezähler gleich 0), als auch die schreibende Sperre nicht gesetzt ist (Schreibzähler gleich 0). Sind diese beiden Voraussetzungen erfüllt, kann der anfragende Prozeß diese Sperre setzen und hat dann die alleinige Schreibberechtigung auf der Indexstruktur. Eine lesende Sperre kann nur gesetzt werden, wenn keine schreibende Sperre gesetzt ist (Schreibzähler gleich 0). Ist dies der Fall, darf der anfragende Prozeß diese Sperre setzen und hat den mit anderen Prozessen, die eine Lesesperre halten, gleichberechtigten, lesenden Zugriff auf diese Indexstruktur.

Näheres zum Verfahren „Write once/Read multiple“ kann z.B. in [ROT99] nachgelesen werden.

4.2 Der kd-Baum

Im folgenden soll der kd-Baum, dessen Aufbau in 2.4.2 besprochen wurde, auf die Erfüllung der Forderungen *SAF1* bis *SAF7* hin überprüft werden. Es wurde beschlossen, den kd-Baum gemäß zweier unterschiedlicher Ansätze zu analysieren und zu implementieren.

Bei der ersten Variante handelt es sich dabei um die denkbar einfachste Implementierung: Jeder Knoten kann genau ein Element, repräsentiert z.B. durch Position und eindeutige Kennung, aufnehmen. Dabei wird für einen Knoten keine Unterscheidung zwischen Blatt und Index durchgeführt, sondern alle Knoten gleich behandelt. Nachfolgerzeiger werden erst bei Einfügen eines passenden Elementes gesetzt, zuvor haben diese den Wert `null`. Kd-Bäume dieser Art werden oft auch als homogene kd-Bäume bezeichnet. Der Name ergibt sich hier aus der Nichtunterscheidung, die zwischen Blatt- und Indexknoten getroffen wird. Durch das Halten genau eines Objektes pro Knoten ergibt sich beispielsweise, daß die Anzahl der im kd-Baum gehaltenen räumlichen Objekte gleich der Anzahl der Knoten ist.

Die zweite Variante unterscheidet Knoten in Indizes und Blätter. Anders als in vielen anderen räumlichen Datenstrukturen, wie z.B. dem Quad-Tree, wird aber auch in den Indizes des Baumes ein Element gehalten. Dieses wird Referenzelement genannt, es dient vor allem dazu, nachfolgende Suchvorgänge zu leiten. Ein Vorteil dieser Variante ist es, daß die Indizes auch dazu benutzt werden, Daten zu halten, was zu einer besseren mitteren Hauptspeicherausnutzung und auch zu einem geringeren Speicheraufwand pro Element führt. Die Blätter dieser Variante eines kd-Baums bieten Platz, eine bei Erzeugung des Baums festlegbare Menge von Elementen aufzunehmen. Nachteile ergeben sich aus den üblicherweise nicht voll besetzten Blättern, die diese positiven Effekte mindern, oder gar ins Negative umkehren können. Entsprechend zum vorher beschriebenen homogenen kd-Baum spricht man hier auch vom heterogenen kd-Baum.

Im Bezug auf die Behandlung der unter 2.4.1 genannten Grundprobleme jeder mehrdimensionalen Indexstruktur läßt sich feststellen, daß der homogene kd-Baum keinerlei Erhaltung der topologischen Struktur bietet. In jedem Knoten wird nur ein Objekt gehalten, dadurch wird jedes Objekt von den anderen isoliert. Im Falle des heterogenen kd-Baums ist zumindest ein gewisses Maß an Topologieerhaltung gegeben, denn die in einem Blatt gespeicherten Objekte liegen auch in der Realität nahe beisammen. Nachteilig ist natürlich auch hier, daß in jedem Knoten ein Element gehalten wird, dies führt wieder zur oben beschriebenen Isolation einzelner Objekte. Ebenso werden Objekte die in der Realität nahe beisammen, aber auf verschiedenen Seiten einer Partition liegen, im kd-Baum unter Umständen in weit voneinander entfernten Blättern abgelegt.

Weiterhin ist der kd-Baum für stark variierende Objektdichten nicht besonders geeignet. Variierende Objektdichten führen bei einem kd-Baum immer dazu, daß manche Nachfolgerzeiger in einem Knoten bevorzugt verfolgt werden, da z.B. in einem Teilgebiet mehr Elemente auf der „linken“ Seite liegen, als auf der „rechten“. Dies führt, durch das Fehlen von Balancierungsmethoden, unweigerlich zu einer ungleichmäßigen Baumstruktur.

Dies stellt die Frage nach den Möglichkeiten, die Reorganisation eines kd-Baums durchzuführen. Es ist beispielsweise denkbar, für einen heterogenen kd-Baum eine Methode zu implementieren, die regelmäßig, oder bei Eintreten eines gewissen Auslösezustandes schlecht besetzte Blätter mit anderen vereinigt und im jeweiligen Vaterindex, der dann zukünftig als Blatt behandelt wird, zu speichern. Ein homogener kd-Baum ist hierfür allerdings nicht geeignet, denn jedes Blatt, wie auch jeder Index, sind mit einem Objekt vollständig ausgelastet. Aus diesem Grund können keine entsprechenden Vereinigungen von Blättern durchgeführt werden.

SAF1: Stabilität bei dynamischem Zugriffsverhalten.
Grundsätzlich kann der kd-Baum, wie jede andere unbalancierte Baumstruktur auch, keine Garantien bezüglich des Laufzeitverhaltens in dynamischen Umgebungen geben. Bei ungünstig aufeinanderfolgenden Einfüge-, Lösch- und Aktualisierungsoperationen mit ebenfalls ungünstigen räumlichen Daten kann eine von der optimal ausbalancierten Baumstruktur deutlich abweichende Baumstruktur entstehen.

Für den kd-Baum im statischen Verwendungskontext, d.h. alle Datenpunkte sind vor Beginn der Operationenfolge bekannt, existiert ein Einfügealgorithmus, der einen balancierten kd-Baum erzeugt und in $O(n \log n)$ Operationen bzw. Vergleichen arbeitet. Dabei wird die gegebene, in den zuerst leeren Baum einzufügende Punktmenge rekursiv so aufgeteilt, daß die durch eine Teilung entstehenden beiden Teilmengen jeweils möglichst gleich viele Mitglieder haben (Wahl des Medians für die jeweilig maßgebende Dimension der einzufügenden Punkte). Dies wird solange fortgeführt, bis sich in jeder der entstandenen Teilbereiche des Universums nur noch eine gewisse, maximale Anzahl Elemente befindet. Diese Anzahl ist gleich der festen Anzahl der in einem Knoten maximal zu haltenden Elemente. In einem dynamischen Kontext ist ein solcher balancierter Aufbau verständlicherweise nicht möglich, denn Einfüge-, Lösch- und Aktualisierungsoperationen treten in einer nicht vorhersehbaren Reihenfolge mit nicht vorhersehbaren räumlichen Datenwerten auf.

Eine Möglichkeit, einen balancierten kd-Baum in einer dynamischen Umgebung zu erzwingen wäre, bei Einfügen eines Elementes, daß zu einer Unbalanciertheit führen würde, eine Neukonstruktion des gesamten kd-Baums durchzuführen, was bei einer dann bekannten Punktmenge in *O(n log n)*-Aufwand durchgeführt werden kann (siehe oben). Dies führt aber natürlich zu einem deutlich erhöhten Zeitaufwand, um den Baum balanciert zu halten, der sich wohl nur in Sonderfällen rechnen wird.
Weiterhin kann diese Möglichkeit bei hoher Dynamik der Zugriffe auf die Indexstruktur voraussichtlich nicht angewendet werden, denn während der Baum neu konstruiert wird, können keine Anfragen oder andere Aktualisierungen auf diesem durchgeführt werden, was für die Antwortzeiten dieser Operationen äußerst ungünstig ist.

Nguyen, DuPrie und Zografou [NGU98] stellen fest, daß für den kd-Baum (sowie den Quad-Tree) im wesentlichen keine bekannten Methoden existieren, um diesen unter dynamischen Bedingungen balanciert zu halten. Die unten stehende Abbildung verdeutlicht noch einmal die starke Abhängigkeit der Struktur eines kd-Baums von der Einfügereihenfolge seiner Objekte.

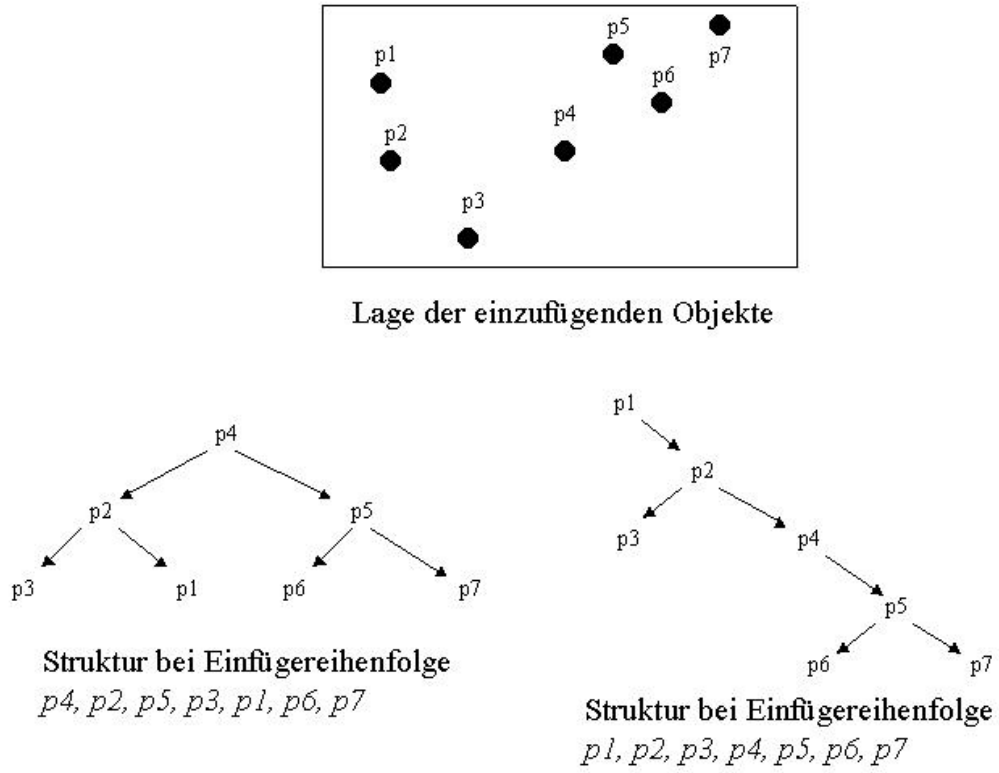


Abbildung 13: Starke Abhängigkeit der Struktur eines kd-Baums von der Einfügereihenfolge seiner Objekte

Da keine empirischen Untersuchungen zur Degenerationsneigung von kd-Bäumen in einer dynamischen Umgebung vorliegen, wird das Verhalten dieser Indexstrukturen für lange andauernde dynamische Zugriffe im nächsten Kapitel experimentell überprüft.

SAF2: Effiziente Ausführung „schreibender" Zugriffe

Bei Einfüge-Operationen muß zunächst einmal die Position im Baum, die ein Element nach dem Einfügen einnehmen soll, gefunden werden. Dies wird beim kd-Baum erreicht, indem von der Wurzel ausgehend für jeden Knoten abhängig von der Tiefe, in der dieser Knoten im Baum liegt, ein Vergleich einer Komponente der räumlichen Daten des dort gespeicherten Elements durchgeführt wird. Ist die jeweilig gewählte Komponente des räumlichen Datums des einzufügenden Elements kleiner als die entsprechende Komponente des in einem Knoten gespeicherten Elements, so wird der „linke" Nachfolge-Pfad verfolgt, im anderen Fall der „rechte".

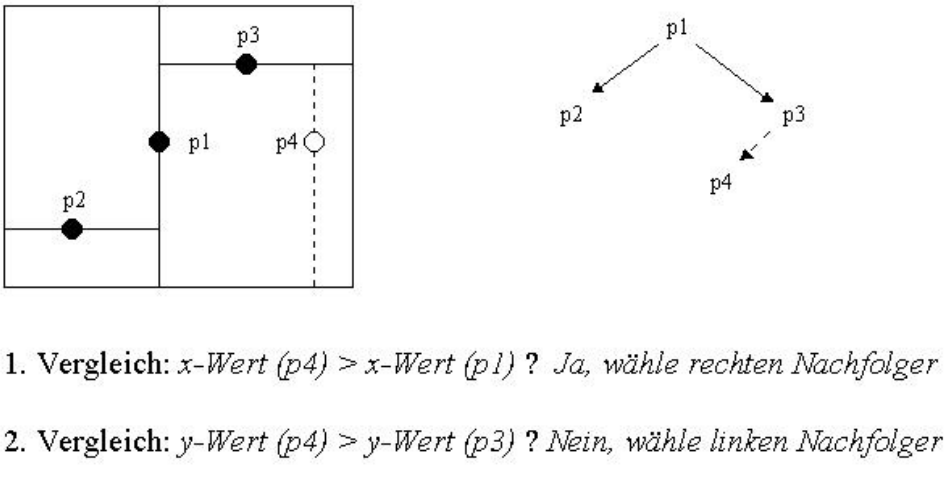


Abbildung 14: Traversierung eines kd-Baums

Ein Vorteil der so gestalteten kd-Baum-Traversierung ist die geringe Anzahl von Vergleichen, die nötig sind, um den richtigen Nachfolgezeiger für ein einzufügendes Element zu bestimmen. Diese erfordert beim kd-Baum nur einen Vergleich für jeden besuchten Knoten. Bei anderen, auf Vollvergleichen basierenden mehrdimensionalen Indexstrukturen werden im allgemeinen bei *d* Dimensionen auch *d* Vergleiche benötigt. Vorteilhaft ist auch, gerade bei der Verwendung im Zusammenhang mit räumlichen Daten, daß statt eines unter Umständen teuren „räumlichen" Vergleiches, wie z.B. der Berechnung der Distanz zweier Punkte in einem geografischen Koordinatensystem, ein einfacher arithmetischer Vergleich zweier Zahlen ausreichend ist („Ist die geografische Länge von Punkt *x* größer oder kleiner als die

geografische Länge von Punkt *y*?“). Eine negative Eigenschaften einer solchen Baumorganisation ist aber der potentiell längere Zugriffspfad auf ein Element. Da beim kd-Baum nur je zwei Nachfolgemöglichkeiten in jedem Knoten zur Verfügung stehen, ist der Verzweigungsfaktor eines kd-Baums kleiner als der Verzweigungsfaktor beispielsweise eines Quad-Trees. Dies führt zu potentiell längeren Pfaden durch den Baum, bis der passende Einfügeort für ein Element gefunden wurde, oder die Daten dieses Elementes ausgelesen werden können.

Das Löschen eines Elementes führt bei einem kd-Baum im schlimmsten Fall dazu, alle Elemente, die sich im Baum unterhalb des zu löschenden Elementes befanden, gleichfalls zu löschen und neu in den Baum einzufügen. Dies ist erforderlich, weil das bei Löschen eines Elementes in anderen Baumstrukturen übliche Vorgehen: „Wähle unter den bisherigen Nachfolgern des gelöschten Elementes einen passenden Nachfolger aus und verwende diesen als neues Referenzelement in diesem Knoten“ nicht ohne weiteres möglich ist.
Dies hat seinen Grund in der Tatsache, daß das Ergebnis eines Vergleiches der Werte einer bestimmten Komponente zweier mehrdimensionaler Daten nicht notwendigerweise mit dem Ergebnis eines Vergleiches der Werte einer anderen Komponente dieser Daten übereinstimmt. Daraus folgert, daß nach dem Löschen eines Elementes und dem Auswählen eines der Nachfolger für alle anderen Elemente überprüft werden muß, ob diese sich „links“ oder „rechts“ dieses Elementes befinden, d.h. ob die Werte ihrer entsprechenden Datenkomponenten kleiner oder größer dem des neuen Referenzelementes ist.
Dieser Vergleich liefert, wie leicht einzusehen ist, im allgemeinen nicht dieselben Ergebnisse wie zuvor. Das bedeutet, daß die zuvor aufgestellte Ordnung im Baum nicht übernommen werden kann und so das oben angesprochene Löschen und Neueinfügen nötig wird. Für das Neueinfügen der Elemente ist die vorherige Sortierung der Einfügekandidaten für jede Einfügeebene im Baum denkbar. Wählt man aus dieser sortierten Menge dann beispielsweise den Median aus, so wird die Aufteilung des Universums in Teilbereiche gleichmäßiger vollzogen, als bei einer willkürlicher Einfügereihenfolge. Zusätzliche Kosten entstehen dann aber durch die Sortierung bzw. das Finden des Medians.
Der homogene kd-Baum ist nicht geeignet, um effizient Aktualisierungsoperationen ausführen zu können. Die Aktualisierung der Position eines Objekts muß so durchgeführt werden, daß zunächst das Objekt (mit alter Position) komplett gelöscht wird und nachfolgend (mit neuer Position) wieder eingefügt wird. Es existiert beim homogenen kd-Baum keine Möglichkeit, zunächst zu überprüfen, ob die Position des zu aktualisierenden Objektes lokal

in einem Blatt geändert werden kann, ohne daß die Sortierungsbestimmungen des kd-Baums verletzt werden, da pro Knoten nur ein Objekt gehalten wird. Dies führt dazu, daß bei jeder Aktualisierung eine Lösch- und eine Einfügeoperation durchgeführt werden muß, wobei aus der Löschoperation wiederum das Löschen und Einfügen vieler anderer Objekte folgen kann.

Für den heterogenen kd-Baum ist es dagegen möglich, bei Aktualisierungsoperationen zunächst überprüfen zu lassen, ob die vorzunehmende Positionsänderung eines Objektes dazu führt, daß dieses Objekt fortan in einem anderen Blatt gehalten werden muß, oder ob eine Aktualisierung der Position in diesem Blatt zulässig ist. Dies erfordert aber das Mitführen von zusätzlichen Informationen in einem Blatt, etwa über das abgedeckte Gebiet, und das fortlaufende Aktualisieren dieser Information bei Einfügen, Löschen usw.. Aus Aufwandsgründen wurde bei der vorliegenden Implementierung des heterogenen kd-Baums allerdings darauf verzichtet und Aktualisierungsoperationen wie beim homogenen kd-Baum ausgeführt.

SAF3: Effizienz bei ungleichmäßigen Datenverteilungen

Wie alle Indexstrukturen, die ohne inhärente Maßnahmen zur Balancierung entworfen wurden, kann sich der kd-Baum nicht an ungleichmäßig verteilte Datensätze (engl.: Skewed Data Sets) anpassen. Die Position eines Objektes im Baum wird schließlich nur von dessen relativer Position zu den anderen, zuvor eingefügten Objekten bestimmt. Dies stellt ein bekanntes Problem bei kd-Bäumen dar, denn eine schlecht balancierte Baumstruktur führt zu potentiell längeren Ausführungszeiten für Anfragen und Aktualisierungsvorgänge. Besonders ist dabei der homogene kd-Baum betroffen, dessen Leistungsfähigkeit in besonderem Maße von einer balancierten Struktur abhängig ist. Dies deshalb, da bei Halten nur eines Elements pro Knoten die benötigte Zeit für Traversierungen proportional zur Anzahl der verfolgten Nachfolgerzeiger ist. Dieses Verhalten wird durch die Ergebnisse einiger der im nächsten Kapitel beschriebenen Experimente noch einmal verdeutlicht.

SAF4: Einfachheit

Der kd-Baum ist in der ursprünglich von Bentley vorgeschlagenen Fassung von der Einfachheit seiner Struktur und Organisation sehr positiv zu bewerten. Gerade die einfache Traversierung, die mit dem bloßen Vergleich zweier Zahlwerte pro Knoten auskommt und die von der Anzahl der Dimensionen unabhängige Anzahl von Nachfolgerzeigern in einem

Knoten machen den kd-Baum zu einer unkomplizierten Datenstruktur, die sich mit wenig Aufwand auch für Anwendungen mit mehr als zwei oder drei Dimensionen erweitern läßt.

SAF5: Effiziente Ausübung der Operationen „Bereichsanfrage“ und „Nächster-Nachbar-Anfrage“

Eine Bereichsanfrage läßt sich mit dem kd-Baum so implementieren, daß ausgehend von der Wurzel jeweils überprüft wird, ob die Position des Referenzelementes eines Knotens im Suchfenster enthalten ist, oder nicht. Ist dies der Fall, so wird das entsprechende Element in die Lösungsmenge dieser Anfrage aufgenommen und die Suche wird rekursiv bei den vorhandenen Nachfolgern fortgesetzt. Im anderen Fall wird überprüft, ob das Suchfenster die in diesem Knoten als Diskriminator dienende Hyperebene schneidet oder nicht. Im ersten Fall wird die Suche wieder rekursiv bei den vorhandenen Nachfolgern fortgesetzt, im anderen Fall nur bei einem der beiden möglichen Nachfolger, nämlich dem, der in dem durch die Hyperebene festgelegten Teilgebiet liegt, in dem sich auch das Suchfenster voll befindet. Die Tiefensuche in einem Teilbaum endet, wenn keine geeigneten Nachfolgerzeiger mehr verfolgt werden können. Sind schließlich alle Untersuchen beendet, wird die gesamte Lösungsmenge ausgegeben. Es existiert für die Bereichsanfragen mit achsenparallelen Suchfenstern in einem kd-Baum eine obere Schranke, $O(n^{\frac{d-1}{d}})$, wobei der Parameter d die Anzahl der Dimensionen des kd-Baums bezeichnet [ERI98]. Für den in dieser Arbeit hauptsächlich betrachteten zweidimensionalen Fall ergibt sich für eine solche Bereichsanfrage die obere Schranke von $O(\sqrt{n})$.

Die bei einer Nächster-Nachbar-Anfrage übliche Vorgehensweise ist es, zuerst rekursiv den Blattknoten aufzusuchen, der den Suchpunkt enthalten würde. Unter den dort gespeicherten Objekten wird der oder die nächstgelegenen Nachbarn ausgewählt. Als nächstes wird überprüft, ob ein Fortsetzen der Suche noch Sinn macht, d.h. ob sich potentiell noch weitere Objekte in anderen Knoten befinden können, die zur Lösungsmenge gehören. Ist dies der Fall, so wird in diesen Knoten gesucht und danach wieder überprüft, ob es noch weitere, nicht gefundene Elemente der Lösungsmenge geben kann. Für das Leistungsverhalten der Anfrage ist es besonders wichtig, die Anzahl der zu untersuchenden Teilbäume schnell einzuschränken, dies wird unter anderem durch das dynamische Aktualisieren des Abstandes des Suchpunktes zum bisher gefundenen nächsten Nachbarn unterstützt. Wenn die Suche endet, enthält die Lösungsmenge dann alle qualifizierten Objekte, bzw. deren eindeutige Bezeichner. Genaueres zu dem hier beschriebenen Algorithmus kann z.B. in [FRI77], oder

[MAR96] nachgelesen werden. In letzterer Quelle ist eine detaillierte Analyse des original von Bentley vorgestellten Algorithmus enthalten.

Eine solche Nächster-Nachbar-Anfrage kann mit einem balancierten kd-Baum in *O (log n)*-Aufwand beantwortet werden ([FRE97], [ERI98]. Für einen nicht balancierten, ja sogar zur linearen Liste entarteten Baum, wie dies aus den weiter oben angegebenen Gründen geschehen kann, ist diese Zeitkomplexität natürlich nicht haltbar.

SAF6: Günstige Speicherausnutzung

Im folgenden sollen die unter 4.1 definierten Bewertungsmetriken zunächst für die erste Variante des kd-Baums dargestellt werden. Die Ermittlung des mittleren Speicheraufwands pro Element, O_{avg}, ergibt sich aus folgenden Überlegungen:

Jeder Baumknoten enthält die folgenden Komponenten:

- Kosten für „linken“ Nachfolger.
- Kosten für „rechten“ Nachfolger.
- Kosten für Höhenangabe innerhalb des Baumes.
- Kosten zur Festlegung der Anzahl von Dimensionen des gespeicherten Elementes.
- Kosten für das Halten der Positionsangabe des gespeicherten Elementes.
- Kosten für das Halten eines eindeutigen Bezeichners des gespeicherten Elementes.

Die Kosten für die Nachfolger und das Halten der Elementdaten sind dabei die hardwareabhängigen Kosten für je einen Zeiger, die Kosten für die Höhen- und Dimensionsangabe, sowie die von der verwendeten Hardware und von der Spezifikation der Implementierungssprache abhängigen Kosten für das Halten zweier natürlicher Zahlen.
In der hier verwendeten Definition der Kosten für das Halten der Elementdaten sind nur die Kosten für das Halten der Zeiger berücksichtigt. Die für das eigentliche Speichern der Elementdaten zusätzlich anfallenden Kosten (im objektorientierten Programmiersprachenparadigma etwa die Kosten, die Daten eines Objektes zu speichern), werden bei dieser Betrachtung außen vor gelassen. Da diese Kosten aber für alle untersuchten Indexstrukturen dieselben sind, reicht die Betrachtung des Speicheraufwandes gemäß obiger Definition für eine Bewertung aus.

Da für den homogenen kd-Baum die Zahl der Knoten mit der Zahl der gehaltenen Elemente gleichgesetzt werden kann (genau ein Element pro Knoten) und aus der oben dargestellten Definition für den Speicheraufwand eines Knotens, ergibt sich der mittlere Speicheraufwand O_{avg} pro Element durch Addition der Kosten der oben dargestellten Komponenten.

Die mittlere Hauptspeicherausnutzung U_{avg} ergibt sich laut oben angegebener Definition durch Division der Größe des „sinnvoll" verwendeten allokierten Speichers durch die Größe des insgesamt allokierten Speichers für alle Knoten. Diese Definition ist für den hier betrachteten homogenen kd-Baum nur bedingt verwendungsfähig, da jeder existente Knoten genau ein Element enthält, und damit den belegten Speicher optimal ausnutzt. Es wurde stattdessen folgende, für diese Variante besser passende Metrik aufgestellt:

Def: U_{avg} für homogenen kd-Baum

- Anzahl der gesetzten Nachfolgerzeiger: n_u
- Gesamtanzahl von Nachfolgerzeigern im Baum: n_t

$$U_{avg} = \frac{n_u}{n_t}$$

Das Nichtsetzen eines Nachfolgerzeigers ist für den homogenen kd-Baum die einzige Möglichkeit, einen einmal allokierten Speicherplatz nicht „sinnvoll" zu verwenden, denn alle anderen für einen neuen Knoten allokierten Speicherplätze werden direkt bei dessen Erzeugung mit Werten besetzt. Unabhängig von der jeweiligen Ausprägung des kd-Baums liefert die Anwendung dieser Metrik für Bäume mit vielen Knoten ein nahezu konstantes Ergebnis (≈ 0,5). Dies ist leicht nachvollziehbar, denn für die Besetzung der Nachfolgerzeiger eines Knotens existieren genau drei Möglichkeiten:

1. Der Knoten ist ein innerer Baumknoten und hat zwei gesetzte Nachfolgerzeiger.
2. Der Knoten ist ein innerer Baumknoten und hat einen gesetzten Nachfolgerzeiger.
3. Der Knoten ist kein innerer Baumknoten und hat demnach keinen gesetzten Nachfolgerzeiger.

Im ersten Fall ist für diesen Knoten kein unnötiger Speicheraufwand gegeben. Durch die beiden gesetzten Nachfolgerzeiger gibt es aber zwei Knoten, die potentiell wieder nichtbesetzte Nachfolgerzeiger besitzen können. Im zweiten Fall führt dieser Knoten zu

einem unnötigen Speicheraufwand von einem nicht gesetzten Zeiger, und es existiert noch ein Knoten, der wiederum nicht gesetzte Zeiger besitzen kann. Im dritten Fall erzeugt dieser Knoten einen unnötigen Speicheraufwand von zwei nicht gesetzten Zeigern. Da dieser aber keine Nachfolger hat, endet die Zeigerkette hier und es existieren keine weiteren Knoten mit potentiell nicht gesetzten Zeigern unter diesem. Betrachtet man nun den Baum von der Wurzel zu den äußeren Knoten, so kann man feststellen, daß ein Knoten mit *u* ungesetzten Zeigern zu *(2-u)*2* unbesetzten Nachfolgerzeigern führen kann, aber selbst *(2-u)* Zeiger setzt. Setzt man diese beiden Werte in das oben definierte Verhältnis, so ergibt sich für die erste Variante des kd-Baums die oben angegebene, für größere Bäume gegen den Wert 0,5 strebende mittlere Hauptspeicherausnutzung U_{avg}.

Die Untersuchungen der mittleren Auslastung der Blätter fallen für den homogenen kd-Baum weg, diese beträgt nach Definition immer 100 %.

Nachfolgend sollen die soeben für die homogene Variante besprochenen Metriken zur Speicherausnutzung nun auch für den heterogenen kd-Baum untersucht werden.

Die Ermittlung des mittleren Speicheraufwands pro Element O_{avg} ergibt sich hier folgendermaßen: Der Speicheraufwand eines Knotens, unabhängig, ob dieser ein Index oder ein Blatt ist, setzt sich zusammen aus:

- Kosten für die Angabe der maximalen Anzahl von Elementen, die ein Blatt enthalten kann (n_{max}).
- Kosten für die Angabe der aktuellen Anzahl von Elementen, die ein Blatt gegenwärtig enthält (n_{curr}).
- Kosten für Höhenangabe innerhalb des Baumes.
- Kosten zur Festlegung der Anzahl von Dimensionen des gespeicherten Elementes.
- Kosten für das Halten der Positionsangaben von nmax Elementen (wird nur genutzt, wenn Knoten Blatt ist).
- Kosten für das Halten der eindeutigen Bezeichners von nmax Elementen (wird nur genutzt, wenn Knoten Blatt ist).
- Kosten für „linken“ Nachfolger.
- Kosten für „rechten“ Nachfolger.

- Kosten für das Halten der Positionsangabe des Referenzelementes (wird nur genutzt, wenn Knoten Index ist).
- Kosten für das Halten des eindeutigen Bezeichners des Referenzelementes (wird nur genutzt, wenn Knoten Index ist).

Vernachlässigbar klein sind dabei die Kosten für die Speicherung einer boolschen Variable pro Knoten, mit deren Hilfe jeweils überprüft werden soll, ob dieser ein Blatt `(true)` oder ein Index `(false)` ist.

Die oben angegebenen Komponenten zur Berechnung des Gesamtspeicheraufwandes O_{tot} eines heterogenen kd-Baums setzen sich nun wie folgt zu diesem zusammen.

Gesamtspeicheraufwand O_{tot}:

- Anzahl der Knoten: n_n
- Summe des allokierten Speichers der einzelnen Komponenten für einen Knoten: O_n

$$O_{tot} = n_n * O_n$$

Und damit ergibt sich für O_{avg} (mit n_e als Anzahl der Elemente):

$$O_{avg} = \frac{O_n * n_n}{n_e}$$

Bei der Betrachtung der mittleren Hauptspeicherausnutzung U_{avg} muß nun, anders als beim homogenen kd-Baum, die Verwendung eines Knotens als Index oder als Blatt mitberücksichtigt werden. Ist dieser ein Index, so wird der bei Erzeugung dieses Knotens allokierte Speicherplatz für die Zeiger auf die n_{max} Positionsangaben und Bezeichner nicht verwendet. Gleichermaßen ist die Angabe der Anzahl der momentan in diesem Knoten gespeicherten Elemente eigentlich nicht erforderlich, genauso wie die Angabe der Maximalanzahl der zu speichernden Elemente.
Im anderen Fall, wenn ein Knoten von der Funktionalität her ein Blatt ist, wird der für die Speicherung der Angaben für die Höhe eines Knotens, der Anzahl der Dimensionen, sowie für die Haltung der Zeiger auf die beiden Nachfolger, die Positionsangabe und den eindeutigen Bezeichner des Referenzelementes allokierte Speicher nicht verwendet.

Damit ergibt sich für U_{avg}:

- Durchschnittlich verwendeter Speicherplatz für einen Index: M_i
- Anzahl der Indizes: n_i
- Durchschnittlich verwendeter Speicherplatz für ein Blatt: M_l
- Anzahl der Blätter: n_l
- Allokierter Speicherplatz für einen Knoten: M_n
- Anzahl der Knoten: $n_n (= n_i + n_l)$

$$U_{avg} = \frac{M_i * n_i + M_l * n_l}{M_n * n_n}$$

Erwähnenswert ist hier auch das Index/Blatt-Verhältnis, das gegen den Wert 1 strebt. Dies ergibt sich aus der Tatsache, daß ein überlaufendes Blatt die Anzahl der Blätter um 1 verringert und die Anzahl der Indizes um 1 erhöht, da es nun als Index statt als Blatt behandelt wird. Gleichzeitig aber erhöht sich die Anzahl der Blätter durch das Neuerzeugen zweier neuer Blätter um 2, womit sich zusammengefaßt eine Steigerung der Blattanzahl um ebenfalls 1 ergibt. Dies aber führt zu obiger Beobachtung.

.

SAF7: Unterstützung von Sperrmechanismen

Für den kd-Baum existieren keine bekannten speziellen Sperrmechanismen zur Synchronisierung konkurrierender, paralleler Zugriffe. Dies führte zur Implementierung des allgemein verwendbaren „Write once/Read multiple“-Verfahrens, das am Anfang dieses Kapitels vorgestellt wurde.

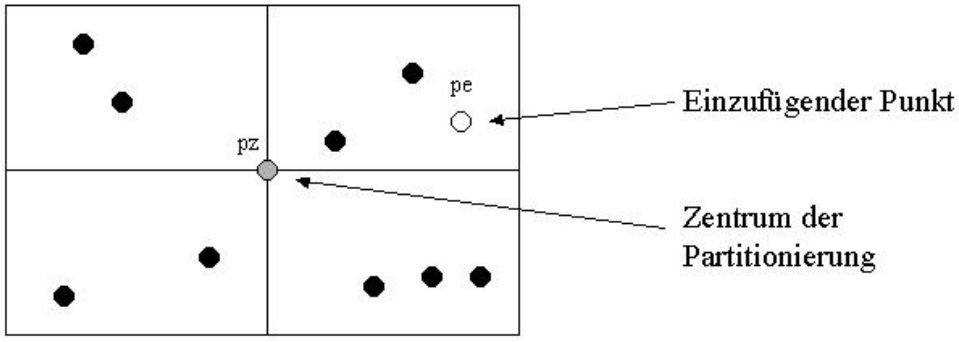
4.3 Der Quad-Tree

In diesem Unterkapitel soll der in Kapitel 2.4.2 vorgestellte Quad-Tree auf seine Eignung gemäß SAF1 bis SAF7 untersucht werden.

Die hier verwendete und vor allem im Rahmen von SAF5 (günstige Speicherausnutzung) genauer analysierte Implementierung basiert auf der von Alexander Leonhardi im Rahmen des Lokationsdienstes der NEXUS-Platttform bereits implementierten Version des Quad-Trees.

Wie auch schon beim zuvor besprochenen heterogenen kd-Baum wird einem einzelnen Knoten, abhängig von der Position im Baum, entweder der Verwendungszweck „Blatt“, oder der Verwendungszweck „Index“ zugeordnet. Unabhängig vom jeweiligen Verwendungszweck hat jeder Knoten jedoch dieselbe Struktur, d.h. er besitzt dieselben Variablen und Methoden wie alle anderen Knoten.

Ein Blatt kann dabei maximal eine bei Erzeugung des Baumes festgelegte Anzahl von Elementen aufnehmen. Indizes enthalten, im Gegensatz zum kd-Baum, keine Elemente. Der Zugriff auf ein Element wird ausgehend von der Wurzel in jedem besuchten Index so geleitet, daß jeweils überprüft wird, in welcher der vier Teilregionen, in die die überdeckte Fläche des Index aufgeteilt wurde, sich das gesuchte Element befindet. Dies geschieht durch Vergleich der Koordinaten des einzufügenden Elementes mit den Koordinaten des im Zentrum der vier Partitionen liegenden Punktes.



1. Vergleich: *x-Wert (pe) > x-Wert (pz) ?* *Ja, Punkt liegt in östlichen Quadranten*

2. Vergleich: *y-Wert (pe) > y-Wert (pz) ?* *Ja, Punkt liegt in nördlichen Quadranten*

Resultat: *Punkt liegt im nordöstlichen Quadranten*

Abbildung 15: Traversierung eines Quad-Trees

Bei der verwendeten Implementierung eines Quad-Trees wurden die beiden unter 2.4.2 beschriebenen Varianten „Point-Quad-Tree“ und „Region-Quad-Tree“ mitberücksichtigt. Die erste der beiden Varianten teilt bei Überlauf eines Blattes den von diesem Knoten abgedeckten Raum in vier möglicherweise ungleichmäßig große, achsenparallele Teilgebiete auf. Im Gegensatz zu dem ursprünglichen Vorschlag, den zum Überlauf führenden Punkt in das Teilungszentrum zu stellen, wurde hier versucht, durch die gezielte Wahl des

Zentrumspunktes zu einer möglichst gleichmäßigen Verteilung der räumlichen Objekte auf die resultierenden vier Teilgebiete zu gelangen.
Die zweite Variante teilt ein Blatt bei Überlauf in vier gleichgroße, ebenfalls achsenparallele Teilgebiete auf, die Schnittlinien laufen hier ebenfalls nicht zwingend durch einen im Baum gehaltenen Punkt.

Zur Einhaltung der unter 2.4.1 angesprochenen allgemeinen Anforderungen an eine Indexstruktur läßt sich feststellen, daß der Quad-Tree nur im Falle einer maximalen Belegung von einem Element pro Knoten keine Erhaltung der topologischen Struktur bietet. In diesem Fall ist der Aufbau, ähnlich wie beim homogenen kd-Baum, auch extrem von der Einfügereihenfolge der gehaltenen Objekte abhängig. Dies wird weiter unten noch einmal angesprochen und vertieft werden. Im Falle einer maximalen Blattkapazität größer 1 ist eine bessere Erhaltung der topologischen Struktur gegeben, da Objekte, die in der Realität in einer bestimmten Partition des Raums liegen, auch in ihrem durch den Quad-Tree erstellten Bildbereich in einer Partition liegen.

Die in 2.4.1 geforderte Eigenschaft der Anpassung an variierende Objektdichten ist beim Quad-Tree durch die rekursive Verfeinerung stärker besetzter Gebiete gegeben, besonders trifft dies auf die Implementierungsvariante „Point-Quad-Tree“ zu, bei der Partitionierungen bei Überlauf eines Blattes flexibel an die vorliegende Verteilung der Objekte dieses Blattes angepaßt werden.

Methoden zur dynamischen Reorganisation werden vom Quad-Tree nicht implizit durchgeführt. Es ist aber möglich, zur Vermeidung schlecht besetzter Blätter beispielsweise nach jedem erfolgreichen Löschvorgang für den Vater des betroffenen Blattes die Gesamtzahl der in dessen Söhnen gehaltenen Objekte festzustellen. Unterschreitet dieser Wert die Maximalanzahl von Objekten, die in einem Blatt gespeichert werden können, so kann der Vater und dessen 4 Söhne zu einem einzigen Blatt zusammengefaßt werden, das dann an die Stelle des bisherigen Vaters tritt. Problematisch im Zusammenhang mit der dynamischen Reorganisation ist das Reorganisieren von Teilbäumen, in dessen Verlauf innere Knoten gelöscht, oder durch eine Umpartitionierung des abgedeckten Gebietes geändert werden können. Das Ändern eines inneren Knotens bringt nämlich unweigerlich die Behandlung aller unter diesem liegenden Knoten, sowie die Neuverteilung aller Objekte, die in diesen Knoten

gehalten werden, mit sich. Für große Bäume mit vielen gespeicherten Objekten bedeutet dies einen erheblichen Mehraufwand.

SAF1: Stabilität bei dynamischem Zugriffsverhalten

Wie schon früher bei der Betrachtung des kd-Baums gesagt, kann eine Baumstruktur ohne inhärente Balancierungsmaßnahmen keine Garantien für gleichbleibende Leistungsfähigkeit unter dynamischen Bedingungen bieten. Es ist prinzipiell immer möglich, daß ungünstige Reihenfolgen von Lösch- und Einfügeoperationen zu einer Degenerierung der Baumstruktur führen. Dies zeigt sich dann in großen Längenunterschieden zwischen dem längsten Pfad durch den Baum und der Länge, die ein Pfad durch den Baum von der Wurzel zu einem Blatt bei einem perfekt ausbalancierten Baum besitzen würde. Allerdings ist die Struktur des Region-Quad-Tree in einem dynamischen Umfeld nicht in dem Maße von der Einfügereihenfolge abhängig, wie dies etwa beim kd-Baum gegeben ist. Dies ist der Fall, da etwa bei einem homogenen kd-Baum das zuerst eingefügte Element zwangsläufig bis zu dessen Löschung aus dem Baum als Referenzelement der Wurzel dient. Ist dieses Element ungünstig zu den nachfolgenden Elementen gelegen, etwa in einer Ecke des abgedeckten Gebietes, so wird in Folge der größte Teil der nachfolgend einzufügenden Elemente beim Einsinkvorgang in den Baum einen bestimmten der beiden Nachfolgerzeiger bevorzugt verfolgen. Auf diese Weise kann sich schnell ein unbalancierter Baum ausbilden. Entsprechendes gilt für den heterogenen kd-Baum, da hier bei Überlauf eines Blattes ein Element als neues Referenzelement ausgewählt wird, und dieses bis zu seiner Löschung als solches dient.

Beim „Region-Quad-Tree“ wird stattdessen bei Überlaufen eines Blattknoten das abgedeckte Gebiet in vier gleich große Teilgebiete partitioniert. Es ist demnach unerheblich, ob das erste Element ungünstig zu den anderen gelegen ist, oder nicht, denn die Aufteilung ist immer regulär und damit unabhängig von den zuvor eingefügten Elementen. Diese Betrachtung gilt allerdings nicht für den Point-Quad-Tree, für den eine ähnliche Abhängigkeit der Struktur von der Einfügereihenfolge der Elemente besteht, wie beim heterogenen kd-Baum.

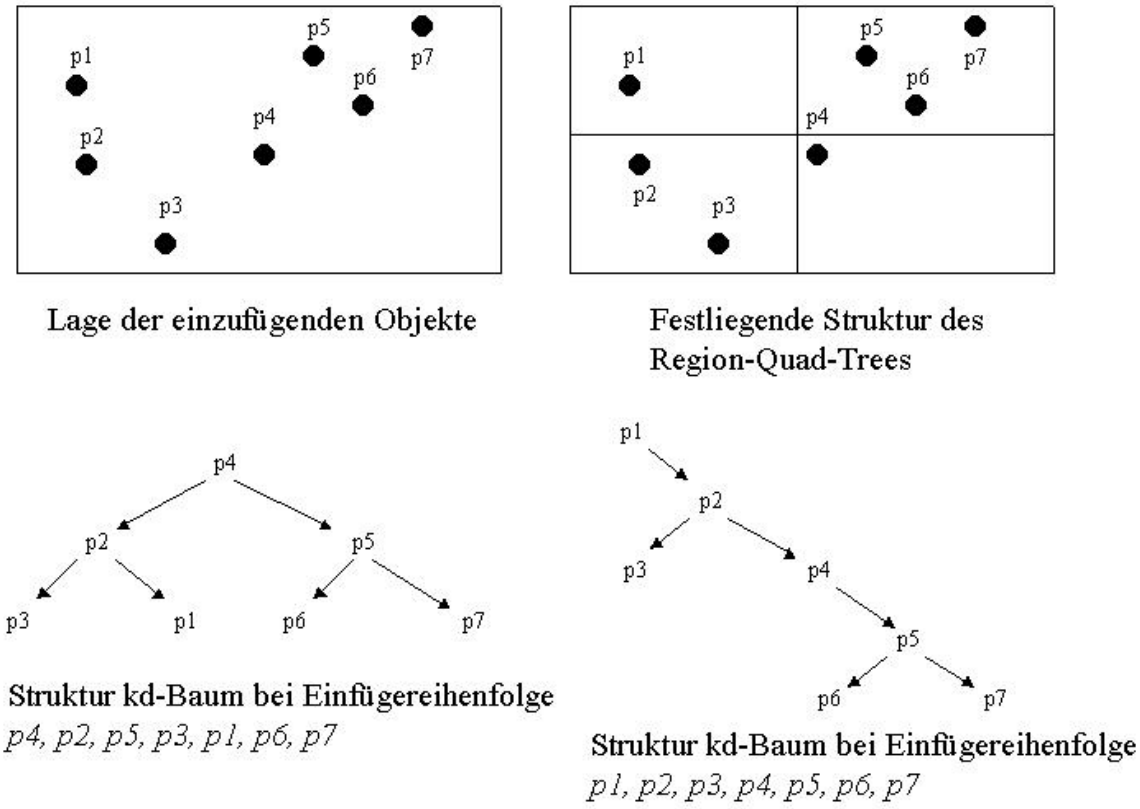


Abbildung 16: Vergleich der Abhängigkeit der Struktur von der Einfügereihenfolge für einen Region-Quad-Tree und einen homogenen kd-Baum

Da auch für den Quad-Tree keine empirischen Untersuchungen für das Verhalten unter dynamischen Bedingungen existieren, wurden auch hier experimentelle Untersuchungen für einen längeren Einsatzzeitraum angestellt. Deren Ergebnisse werden im nächsten Kapitel dargestellt.

SAF2: Effiziente Ausführung „schreibender" Zugriffe

Wie bei vielen zwischen inneren Knoten und Blattknoten unterscheidenden Baumstrukturen beginnt der Einfügevorgang für ein neues Element an der Wurzel des Baumes und endet bei einem Blatt. Innerhalb eines inneren Knotens, oder Indexes, wird jeweils überprüft, in welchem der vier möglichen Teilgebiete der einzufügende Punkt liegen muß und darauf der entsprechende Nachfolgezeiger verfolgt. Innerhalb eines Blattes wird das einzufügende Objekt, wenn noch Speicherplatz frei ist, in die Liste der dort gehaltenen Punkte eingereiht, ansonsten muß eine Aufteilung des betreffenden Blattes in vier neue Teilgebiete, die dann ihrerseits Blätter sind, durchgeführt. Der übergelaufene Blattknoten wird ab diesem Zeitpunkt dann als Index behandelt.

Die oben angesprochene Überprüfung, in welchem Quadrant das einzufügende Objekt liegt, ist für einen Quad-Tree teurer, als etwa bei einem kd-Baum. Dies ist der Fall, da im Gegensatz zu einem kd-Baum pro besuchtem Index *k* Vergleiche bei *k* Dimensionen durchgeführt werden müssen, um den richtigen Nachfolgezeiger auszuwählen. Ein kd-Baum kommt dagegen mit konstant einem Vergleich pro besuchtem Index aus, um mit dem Bestimmen der Einfügeposition fortfahren zu können.

Dies soll noch einmal an einem kleinen Beispiel veranschaulicht werden:

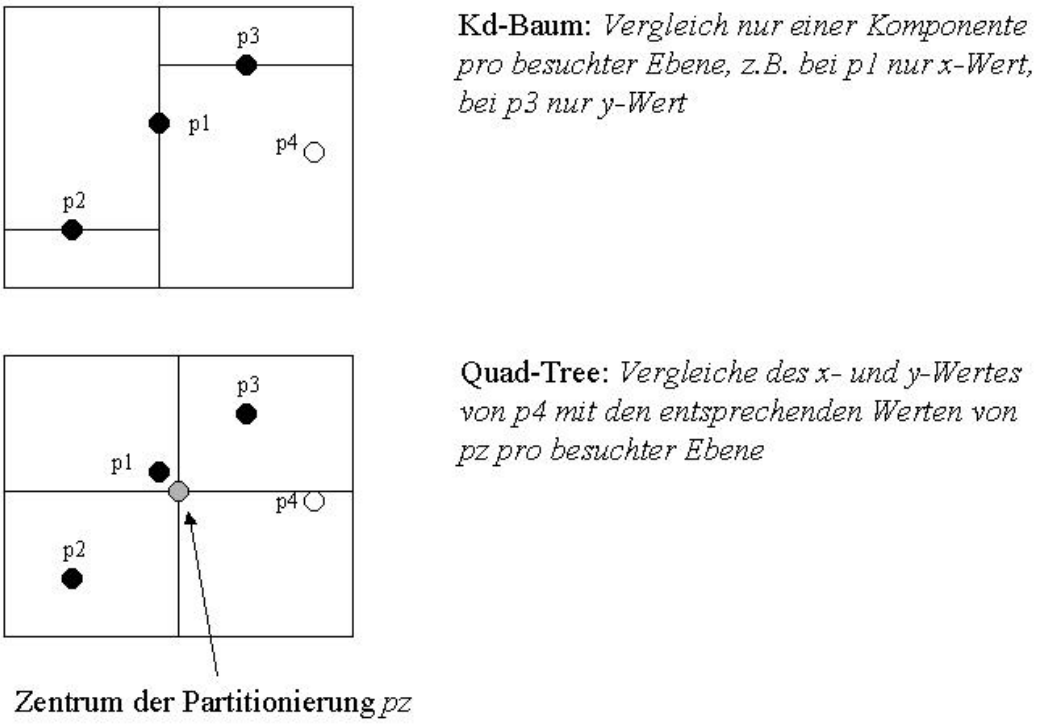


Abbildung 17: Für die Baumtraversierung in jedem Knoten nötige Vergleiche für einen Quad-Tree und einen kd-Baum

Das Auffinden des geeigneten Blattes vor dem eigentlichen Einfügevorgang erfordert bei einem Quad-Tree im allgemeinen allerdings das Verfolgen weniger Zeiger, als bei einem kd-Baum. Dies liegt am größeren Verzweigungsfaktor den der Quad-Tree gegenüber dem kd-Baum besitzt. Das Verfolgen weniger Zeiger führt, wie leicht einzusehen ist, zu weniger besuchten Knoten und damit wiederum zu weniger erforderlichen Vergleichen.

Beim Löschen eines räumlichen Objektes aus einem Quad-Tree wird ein ähnlicher Weg beschritten. Nachdem zunächst von der Wurzel ausgehend jenes Blatt gefunden wurde, in dem das zu löschende räumliche Objekt bei Existenz gehalten wird, werden die in diesem

Blatt gehaltenen Objekte auf exakte Übereinstimmung mit dem ersteren überprüft. Wird eines der Blattelemente als das zu löschende identifiziert, so wird dieses aus der Menge entfernt. Wird ein solches Element nicht gefunden, wird dies als Funktionsergebnis gemeldet und der Löschvorgang abgebrochen. Hier gelten dieselben Überlegungen wie für den Einfügefall, d.h. Löschvorgänge sind pro besuchten Knoten betrachtet aufwendiger als etwa beim kd-Baum, diese Tatsache wird allerdings durch die geringere Anzahl besuchter Knoten wieder relativiert.

Aktualisierungsoperationen werden beim Quad-Tree in der vorliegenden Implementierung so vorgenommen, daß zunächst überprüft wird, ob sich die Aktualisierung der Position eines Objektes lokal auf das Blatt beschränkt, in dem dieses gehalten wurde, oder ob sie zu einer Neupositionierung des Objektes in ein anderes Blatt führen würde. Dazu enthält jeder Knoten einen Verweis auf das von diesem Knoten überdeckte Gebiet. Die oben gestellte Frage ist dann einfach mit einer Enthaltenseins-Anfrage beantwortet, wobei als Eingabe dieser die neue Position des Objektes und das bislang abgedeckte Gebiet des Blattes dient. Auf diese Weise ist ein effizienteres Ausführen von Operationen vom Typ „Aktualisieren“ möglich, als durch das bloße Löschen und Neueinfügen des betreffenden Objektes. Das Halten eines Verweises auf das abgedeckte Gebiet erfordert natürlich das fortwährende Aktualisieren dieses Gebietes bei allen Einfüge-, Lösch- und Aktualisierungsoperationen in diesem Blatt, was wiederum zu Leistungseinbußen bei diesen Operationen führen kann.

SAF3: Effizienz bei ungleichmäßigen Datenverteilungen

Auch der Quad-Tree kann sich nicht an ungleichmäßige Datenverteilungen anpassen, anders gesagt, wurden keine derartigen Maßnahmen bei dessen Entwurf berücksichtigt. Die Position, die ein Element im Baum einnimmt, hängt nur ab von seiner relativen Lage zu den vorher eingefügten Elementen. Wie schon beim kd-Baum führt dies dann zu steigender Ausführungszeit von Anfragen und Aktualisierungen. Vor allem der Variante „Region-Quad-Tree“, die keine Anpassung an die tatsächliche Objektverteilung vornimmt und nur reguläre Partitionierungen zuläßt, kann für diesen Gesichtspunkt nur eine schlechte Bewertung gegeben werden.

SAF4: Einfachheit

Ähnlich wie der kd-Baum ist der Quad-Tree eine gut verständliche, logisch leicht nachvollziehbare Indexstruktur zur Speicherung mehrdimensionaler Daten. Der Verzicht auf

Sonderfälle und die einfach rekursiv implementierbare Baumtraversierung, die ja wie oben bereits angesprochen, sowohl bei Einfüge- und Löschvorgängen, als auch bei Suchvorgängen innerhalb des Baumes verwendet wird, trugen mit zu dessen weiter Verbreitung in vielen Anwendungsgebieten mehrdimensionaler Daten bei.

Vorteile gegenüber dem kd-Baum im Bezug auf das einfache Verständnis hat der Quad-Tree vor allem bei der intuitiv einsichtigeren Traversierung des Baums. Objekte, die sich nordwestlich eines bestimmten Punktes befinden, werden im nordwestlichen Teilgebiet, Objekte, die sich südöstlich dieses Punktes befinden, südöstlich gesucht usw.. Diese Methode der Einteilung einer Menge von Punkten ist dem Anwender intuitiv klarer, als die bei kd-Bäumen vorgenommene Unterscheidung nach der Komponente einer bestimmten Dimension. Nachteile besitzt der Quad-Tree in der etwas komplexeren Implementierung der Traversierung, bei der pro besuchten Knoten mindestens doppelt so viele Vergleiche nötig sind, als bei einem kd-Baum.

SAF5: Effiziente Ausübung der Operationen „Bereichsanfrage“ und „Nächster-Nachbar-Anfrage“

Für einen Quad-Tree gestaltet sich nur das Beantworten von exakten (Punkt-)Anfragen einfach, da auf diese Weise ein eindeutig festgelegter Pfad durch den Baum verfolgt werden kann. Das Ausführen aller anderer, komplexerer Anfragen, wird nicht direkt von der Baumstruktur unterstützt.

Das Beantworten von „Bereichsanfragen“ wird in einem Quad-Tree so durchgeführt, daß ausgehend von der Wurzel der Baum wieder in einer Tiefentraversierung durchlaufen wird. In jedem im Zuge dieser Traversierung besuchten inneren Knoten wird geprüft, mit welcher der vier Partitionen des Raumes das Suchgebiet Schnittpunkte hat. Ist dies für eine Partition der Fall, so wird im entsprechenden Teilbaum weitergesucht. Bei Erreichen eines Blattes werden die dort gehaltenen Objekte auf Enthaltensein im Suchgebiet überprüft, und bei positivem Ergebnis in die Lösungsmenge übernommen. Leider läßt sich durch die potentielle Unbalanciertheit für die Bereichsanfrage keine logarithmische Obergrenze bestimmen.

Für das Beantworten einer „Nächster-Nachbar-Anfrage“ wird im Baum rekursiv im Rahmen einer gewöhnlichen Tiefensuche von der Wurzel zu dem den Suchpunkt enthaltenden Blatt hin abgestiegen. Ist dieses Blatt gefunden, so wird initial der dort gehaltene nächste Nachbar

zum Suchpunkt bestimmt. Nachfolgend wird wie beim kd-Baum die Suche auf die umliegenden Blätter ausgedehnt, wobei vor Aufsuchen eines anderen Teilbaums jeweils überprüft wird, ob dieser einen noch „näheren Nachbarn“ enthalten kann. Üblicherweise werden hierbei dynamisch die momentan besten Ergebnisse bei jedem Schritt bestimmt, so daß am Ende der Suche die Lösungsmenge direkt zurückgegeben und verwendet werden kann. Für einen balancierten Quad-Tree ist wegen der Ähnlichkeit zum kd-Baum auch für eine Nächste-Nachbar-Anfrage eine Implementierung dieser in *O(log n)* möglich, für stark unbalancierte Bäume läßt sich auch hier diese Schranke nicht halten.

SAF6: Günstige Speicherausnutzung

Der Speicherbedarf, der sich pro Knoten ergibt, setzt sich typischerweise zusammen aus den nachfolgend aufgelisteten Komponenten, unabhängig ob ein Knoten funktional als Index oder als Blatt dient (hier ohne Zeiger auf abgedecktes Gebiet für effiziente Aktualisierungen).

- Kosten für die Angabe der maximalen Anzahl von Elementen, die ein Blatt enthalten kann, im folgenden mit n_{max} bezeichnet.
- Kosten für die Angabe der aktuellen Anzahl von Elementen, die ein Blatt gegenwärtig enthält, im folgenden mit n_{curr} bezeichnet.
- Kosten für das Halten der Positionsangaben von n_{max} Elementen (wird nur genutzt, wenn Knoten Blatt ist).
- Kosten für das Halten der eindeutigen Bezeichners von n_{max} Elementen (wird nur genutzt, wenn Knoten Blatt ist).
- Kosten für „nordwestlichen“ Nachfolger.
- Kosten für „nordöstlichen“ Nachfolger.
- Kosten für „südwestlichen“ Nachfolger.
- Kosten für „südöstlichen“ Nachfolger.
- Kosten für das Halten der Positionsangabe des im Zentrum einer Teilung stehenden Elementes für Vergleiche bei nachfolgenden Baumtraversierungen.

Für den Quad-Tree ergibt sich also für O_{avg}, den durchschnittlichen Speicheraufwand pro Element:

- Speicheraufwand pro Knoten: O_n
- Anzahl der Knoten: n_n

$$O_{avg} = \frac{O_n}{n_n}$$

Die mittlere Hauptspeicherausnutzung U_{avg} eines Quad-Trees ergibt sich laut Definition aus der Division der Größe des insgesamt „sinnvoll“ belegten Speichers durch die Größe des insgesamt allokierten Speichers. Für den Betrag des Divisors wird im folgenden der Betrag eingesetzt, der sich durch Multiplikation der Gesamtanzahl der Knoten mit dem Speicheraufwand pro Knoten ergibt. Dies geschieht aus der Überlegung heraus, daß ja für jeden Knoten, unabhängig von dessen funktionalem Zweck, gleich viel Speicherplatz allokiert wird. Als Dividend wird die Summe aus der pro Knoten festzustellenden tatsächlichen Besetzung eines Knotens herangezogen.

Das Index/Blatt-Verhältnis strebt beim Quad-Tree immer gegen den Wert $\frac{1}{3}$. Dies ist leicht verständlich, wenn man sich die Vorgänge beim Überlauf eines Blattes noch einmal vor Augen führt: Ein überlaufendes Blatt vergrößert die Anzahl der Indizes um 1 und verringert die Anzahl der Blätter um 1, da es nun als Index behandelt wird. Gleichzeitig aber entstehen 4 neue Blätter. Damit vergrößert sich die Anzahl der Blätter bei einem Überlauf insgesamt um 3, bei einer gleichzeitigen Vergrößerung der Indexanzahl um 1, was zum oben genannten Ergebnis führt.

SAF7: Unterstützung von Sperrmechanismen

Es existieren für den Quad-Tree keine speziellen Synchronisierungsverfahren, d.h. es gibt keine Synchronisierungsverfahren, die genau auf die Struktur des Quad-Tree abgestimmt sind und daraus Leistungsvorteile ziehen könnten. Stattdessen wurde beschlossen, den im Rahmen der Synchronisierung des kd-Baums vorgestellten „Write once/Read multiple“ zu implementieren. Die Granularität der Sperren war wie beim kd-Baum auf Operationsebene. So war es war z.B. einer schreibenden Operation (Einfügen, Löschen, Aktualisieren) nur dann möglich, ausgeführt zu werden, wenn gleichzeitig keine andere schreibende oder lesende Operation ausgeführt wurde usw..

4.4 Das Grid-File

Die dritte, im Rahmen dieser Arbeit implementierte räumliche Indexstruktur, das Grid-File soll nun auf deren Eignung als Datenhaltungskomponente für den Lokationsdienst der NEXUS-Plattform empirisch untersucht werden.

Die vorgenommene Implementierung richtete sich an der Vorgabe der Diplomarbeit aus, nach der die verwendete Indexstruktur vollständig im Hauptspeicher resident sein sollte. Es wurde dementsprechend also eine rein hauptspeicherbasierte Version des Grid-Files entwickelt, diese verzichtet gänzlich auf Maßnahmen zur Verwaltung von Sekundärspeicher. Für die Bewertung des Grid-Files ist dies eine ungünstige Entscheidung, denn gerade die Garantie, nur zwei Plattenzugriffe für den Zugriff auf ein beliebiges gespeichertes Objekt zu benötigen, ist einer der großen Vorteile dieser Indexstruktur (siehe Kapitel 2.4.3).

Zur Umsetzung der unter 2.4.1 angegebenen grundlegenden Forderungen an eine mehrdimensionale Indexstruktur läßt sich sagen, daß vor allem die Erhaltung der topologischen Struktur der Datenobjekte gut gegeben ist. Objekte, die in der Realität nahe beisammen liegen, werden in ihrer Repräsentation im Grid-File in den meisten Fällen entweder in derselben Zelle, oder in einer der sie umgebenden Zellen gehalten. Dies kann man anschaulich erklären, wenn man sich vorstellt, wie das Grid-File den zur Verfügung stehenden Raum in Zellen aufteilt: Solange die Anzahl der Elemente in einer Zelle unter einer gewissen Schwelle bleibt, werden alle Elemente, die sich in der Realität innerhalb der Grenzen dieser Zelle befinden, in diese Zelle eingefügt. Erst wenn die Schwellenanzahl einmal überschritten wird, wird die betroffene Zelle (und alle anderen durch das Einfügen einer neuen Skala geschnittenen Zellen) geteilt. Objekte, die in der Realität nahe zusammen liegen, sind also nur dann in verschiedenen Zellen untergebracht, wenn sie auf verschiedenen Seiten eines Skalenwertes liegen. Für den Fall der Mitberücksichtigung der Sekundärspeicherverwaltung ist die Erhaltung der topologischen Struktur noch mehr erfüllt, denn mehrere, nebeneinanderliegende Zellen können dann einer Speicherseite zugeordnet werden und so die vorgenommene, möglicherweise ungünstige Plazierung der Skalen teilweise wieder aufheben.

Die zweite der unter 2.4.1 angegebenen Forderungen betrifft die Anpassungsfähigkeit einer Indexstruktur an die möglicherweise variierende Objektdichte im Verwaltungsgebiet dieser.

Hier muß dem Grid-File eine schlechtere Bewertung gegeben werden. Zwar ist eine gewisse Anpassungsfähigkeit an unterschiedliche Objektverteilungen gegeben, da die verschiedenen Zellen nicht dieselben Abmessungen haben müssen. Vielmehr werden bei hoher Objektdichte in einem Teilgebiet die Skalen enger zueinander positioniert werden, während in schwach besiedelten Gebieten die Skalen einen entsprechend größeren Abstand besitzen. Der große Nachteil ergibt sich hier aber aus der Eigenschaft, daß die Partitionierung einer Zelle zu Partitionierungen von unter Umständen vielen Zellen des Rasterverzeichnisses führt. Wenn also wegen der höheren Objektdichte in einem Teilbereich eine feinere Zellpartitionierung nötig ist, so führt dies zur Partitionierung vieler Zellen in anderen Gebieten, die dünner mit Objekten besetzt sind und damit zum Entstehen von vielen eigentlich nicht notwendigen Zellen, die oft sogar völlig leer sind. Für das Grid-File bedeutet dies, daß eine Anpassung an die Objektdichte in einem Teilgebiet zwar gut durchgeführt werden kann, aber dies auch zu einer unnötig feinen Partitionierung in anderen Teilgebieten führt.

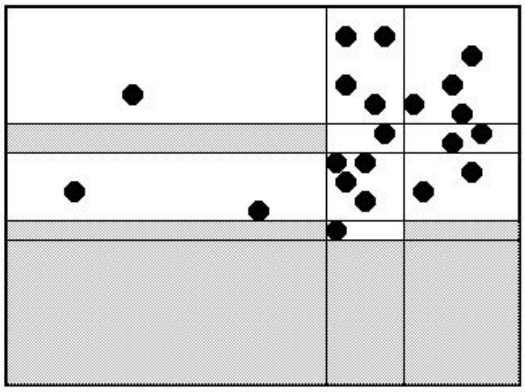
Der Dritte unter 2.4.1 angegebene Punkt, dem Beachtung geschenkt werden muß, ist die Durchführung dynamischer Reorganisationsmaßnahmen. Das Grid-File, genauer das Rasterverzeichnis des Grid-Files, wächst durch die nicht lokalen Zellteilungsvorgänge stark an, dies stellt eines der größten der Probleme des Grid-Files dar, wie nachfolgend noch genauer dargestellt wird. Dadurch ist es für eine konkurrenzfähige Speicherbelegung wichtig, Reorganisationsmethoden anzubieten. In [NIE84] wird vorgeschlagen, durch Mischvorgänge bei schwacher Speicherseiten-Belegung sogenannte *Buddies*, also Zellen, die durch Teilung auseinander hervorgingen, wieder zu einer Zelle zu vereinigen. Dadurch wird dann eine bessere Speicherausnutzung erreicht, da nun mehr Elemente in einer Zelle (bzw. in einer Speicherseite) gehalten werden. Von verschiedener Seite wurde aber auch schon die Rekombination von Zellen vorschlagen, die zuvor nicht notwendigerweise *Buddies* waren, was eine größere Kombinationsauswahl mit sich führt.

Da die im Rahmen dieser Arbeit implementierte Version des Grid-Files aus den oben dargestellten Gründen rein hauptspeicherbasiert war und aus Aufwandsgründen wurde auf Rekombinationsmaßnahmen von Zellen bzw. Speicherseiten ganz verzichtet. Dies führt bei den späteren Betrachtungen zu einer schlechteren Hauptspeicherausnutzung, da nun schwach besetzte Zellen nicht rekombiniert werden können. Stattdessen wurden für das Grid-File Methoden zur Löschung ganzer Zeilen bzw. Spalten leerer Zellen aus der Matrix des Rasterverzeichnisses implementiert, um so die Gesamtzahl der Zellen zu verringern. Zum

einen handelt es sich dabei um Methoden, die entweder eine gesamte Zeile, oder eine gesamte Spalte mit ausschließlich leeren Zellen, jeweils unter Angabe des Zellen- bzw. Spaltenindex, löschen sollen. Diese Methoden können verständlicherweise nur dann erfolgreich ausgeführt werden, wenn die betreffende Zeile bzw. Spalte auch wirklich vollständig aus leeren Zellen bestand, sonst wird die Ausführung der jeweiligen Methode mit negativem Ergebnis abgebrochen. Zum anderen wurde eine Methode implementiert, die alle möglichen Zeilen und Spalten auf vollständig leere Zellen als Mitglieder untersucht und diese nachfolgend löscht, ohne daß die explizite Angabe eines Zeilen- oder Spaltenindex nötig wäre.

SAF1: Stabilität bei dynamischem Zugriffsverhalten

Da das Grid-File, anders als die beiden schon analysierten Indexstrukturen, kd-Baum und Quad-Tree, keine Baumstruktur ist, kann man in diesem Zusammenhang nicht von einem Balancierungsproblem sprechen. Vielmehr ist das große Problem, daß dem Grid-File zugrunde liegt, die nicht lokalen Zellteilungsvorgänge. Erfordert das Einfügen eines neuen Elementes das Partitionieren der betreffenden Zelle, da die Maximalzahl der pro Zelle vorgesehenen Elemente durch Einfügen dieses überschritten wurde, so bewirkt diese Partitionierung auch die Partitionierung all der anderen Zellen, die von der entsprechenden Partitionierungsskala geschnitten werden.



Grau markierte Flächen: Durch nicht lokales Splitverhalten unnötig erzeugte Zellen.

Abbildung 18: Nicht lokales Splitverhalten im Rasterverzeichnis des Grid-Files

Dies führt zu einem exponentiellen Anwachsen der Größe des Rasterverzeichnisses bei Partitionierungsvorgängen. Aus verständlichen Gründen ist dies für eine rein hauptspeicherbasierte Lösung durch die Begrenztheit des Speicherplatzes sehr ungünstig. Schiefe Belegungen, also Belegungen, deren Objektdichte über den zur Verfügung stehenden

Raum stark variiert, führen durch die von den stark besiedelten Gebieten ausgehenden Partitionierungsvorgängen noch zu einer Verstärkung dieses Problems.

Regnier [REG85] zeigt, daß selbst für gleichförmige Objektverteilungen das Wachstum des Rasterverzeichnisses in $O(N^{1+\frac{k-1}{k*b}})$ liegt, für nicht gleichförmige Objektverteilungen kann das Rasterverzeichnis in $O(N^{k})$ anwachsen, wobei *N* die Anzahl der Elemente, *k* die Anzahl der Dimensionen und *b* das Fassungsvermögen einer Speicherseite bezeichnet. Bei Verwendung von Sekundärspeicher schlagen solche Verteilungen aber nicht übermäßig auf die Belegung der Speicherseiten durch, da ja eine *n:1*-Beziehung zwischen Zellen und Speicherseiten vorliegt. Da aber der hier betrachteten Anwendungsfall rein hauptspeicherbasiert ist, stellt das überlineare Wachstum des Rasterverzeichnisses sehr wohl ein praktisches Problem dar.

SAF2: Effiziente Ausführung „schreibender" Zugriffe

Schreibenden Zugriffen liegt das Auffinden einer bestimmten Zelle des Rasterverzeichnisses zugrunde. Einfügeoperationen werden beispielsweise so durchgeführt, daß zuerst die Zelle im Rasterverzeichnis gefunden werden muß, in der das einzufügende Objekt gemäß seiner Position zu plazieren ist. Hat die gefundene Zelle noch Platz, um dieses aufzunehmen, so ist die Operation schon beendet. Ist dies nicht der Fall, so muß ein Partitionierungsvorgang ausgeführt werden, der da nicht von lokaler Natur, relativ aufwendig durchzuführen ist („Teile alle Zellen gemäß der gewählten Skala in zwei neue Zellen, die von dieser Skala geschnitten werden, und passe danach das Rasterverzeichnis an").

Bei Löschvorgängen wird ähnlich vorgegangen, indem zunächst die Zelle gesucht wird, die das zu löschende Objekt enthalten muß. Wird es in dieser Zelle auch aufgefunden, so wird es aus der Zelle entfernt und der Vorgang nach dem Durchführen eventueller Reorganisationsmaßnahmen mit positivem Ergebnis beendet. Bei Implementierung von Mischvorgängen von Zellen, um ein gewisses Maß an Speicherausnutzung zu erreichen, ist nun noch die Überprüfung der betroffenen und der umliegenden Zellen auf das eventuelle Erfüllen eines Auslösekriteriums und das nachfolgende Ausführen der Mischvorgänge nötig. Im anderen Fall wird die Suche mit negativem Ergebnis abgebrochen.

Günstig lassen sich beim Grid-File die schon öfters angesprochenen Maßnahmen zum effizienten Ausführen von Aktualisierungen realisieren, da auf das Halten eines Zeigers um

das aktuell abgedeckte Gebiet einer Zelle zu spezifizieren, verzichtet werden kann. Der Grund hierfür ist, daß der Aufbau des Rasterverzeichnisses schon alle diesbezüglich notwendigen Informationen liefert. Um festzustellen, ob ein Objekt nach durchgeführter Positionsaktualisierung immer noch in derselben Zelle wie vorher gespeichert werden muß, ist lediglich das Ergebnis einer Enthaltenseinsanfrage nötig. Diese erhält als Parameter das von den Skalen des Rasterverzeichnisses spezifizierte Gebiet und die neue Position des zu aktualisierenden Objektes. Liefert die Anfrage ein positives Ergebnis, so kann die alte Position des Objektes einfach gegen die neue Position ausgetauscht werden, im anderen Fall ist das Löschen und nachfolgende Einfügen (mit neuer Position) des Objektes nötig. Das sonst ebenfalls übliche leistungssenkende Aktualisieren des abgedeckten Gebiets pro Zelle/Blatt entfällt für das Grid-File ebenso, da die Skalen bei jedem relevanten Vorgang dieser Art ohnehin auf den neuesten Stand gebracht werden.

SAF3: Effizienz bei ungleichmäßigen Datenverteilungen

Naiv betrachtet könnte man feststellen, daß sich das Grid-File gut an ungleichmäßige Objektverteilungen anpasst. Da Skalen je nach Bedarf eingefügt werden können und die daraus resultierende Partition einer Zelle damit nicht notwendigerweise regulär sein muß, paßt sich das Grid-File bei nur lokaler Betrachtung der betreffenden Zelle recht gut an solche Verteilungen an. Betrachtet man aber die von dieser Zellteilung ausgehenden Teilungsvorgänge für das gesamte Rasterverzeichnis, so führen Häufungen von Elementen in einem bestimmten Bereich des abgedeckten Gebietes zu einer Vielzahl von schlecht besetzten Zellen in den anderen, dünner besiedelten Gebieten. Daraus resultiert auch für ungünstige Objektverteilungen wieder das überlineare Wachstum des Rasterverzeichnisses, welches der Grund für sinkende Effizienz in der Bearbeitung von Anfragen und Aktualisierungen ist. Aus diesem Grund kann auch beim Grid-File nicht von einer guten Anpaßbarkeit an ungleichmäßige Objektverteilungen besprochen werden.

SAF4: Einfachheit

Das Ausführungsprinzip des Grid-Files, mit Hilfe der Dimensionsverfeinerung durch Aufprägen einer Struktur von isoorientierten (rechtwinkligen) Zellen den zur Verfügung stehenden Raum zu partitionieren, ist an sich einfach und intuitiv zu verstehen. Allerdings sind es die relativ vielen Sonderfälle, die eine Implementierung bzw. Erweiterung komplexer werden lassen. Die Verwaltung des Rasterverzeichnisses erfordert beispielsweise von einem Anwender durch die Existenz einiger zu beachtender Besonderheiten (nicht lokale

Partitionierung, Mischen von *Buddies*, ...) ein größeres Maß an Einarbeitungs- und Verständniszeit wie z.B. ein Vertreter der einfachen, baumförmigen Indexstrukturen, wie z.B. dem kd-Baum oder dem Quad-Tree. Ein weiterer Grund für die komplexere Struktur des Grid-Files im Gegensatz zu den beiden gerade genannten Indexstrukturen ist auch die eigentliche Konzeption als PAM, d.h. das eigentliche Vorhandensein von eingebauten Mechanismen zur effektiven Nutzung des Sekundärspeichers. Obwohl diese Mechanismen vorgabenbedingt nicht berücksichtigt wurden, führt ihr Vorhandensein in der Konzeptionsphase des Grid-Files zu einer komplexeren Struktur, als etwa der des hauptspeicherbasierten kd-Baums.

SAF5: Effiziente Ausübung der Operationen „Bereichsanfrage“ und „Nächster-Nachbar-Anfrage“

Für eine Bereichsanfrage kann beim Grid-File besonders unkompliziert die Menge der Zellen bestimmt werden, die mit dem Suchfenster zumindest einen Punkt gemeinsam haben und daher potentiell Kandidaten der Lösungsmenge enthalten können. So werden zuerst die Zellen bestimmt, die z.B. den nordwestlichen und den südöstlichen Begrenzungspunkt des Suchfensters enthalten. Die so gefundenden Zellen begrenzen die vom Suchfenster berührte Menge von Zellen entsprechend als nordwestliche bzw. südöstliche Ecken. Nach Feststellen der Indizes jeder dieser Zellen in jeder Dimension kann der Rest der noch zu untersuchenden Zellen bestimmt werden, deren Index komponentenweise zwischen den vorher bestimmten entsprechenden Komponenten der Eckindizes liegt. Nachdem alle in Frage kommenden Zellen gesammelt wurden, müssen noch alle in diesen Zellen gehaltene Objekte auf Enthaltensein im Suchfenster überprüft werden. Die Überprüfung braucht dabei nur für die äußersten Zellen der gefundenen Menge durchgeführt werden, da alle anderen, weiter innen liegenden Zellen auf jeden Fall komplett vom Suchfenster überdeckt werden und damit alle in ihnen gehaltenen Objekte zur Lösungsmenge der Bereichsanfrage gehören.

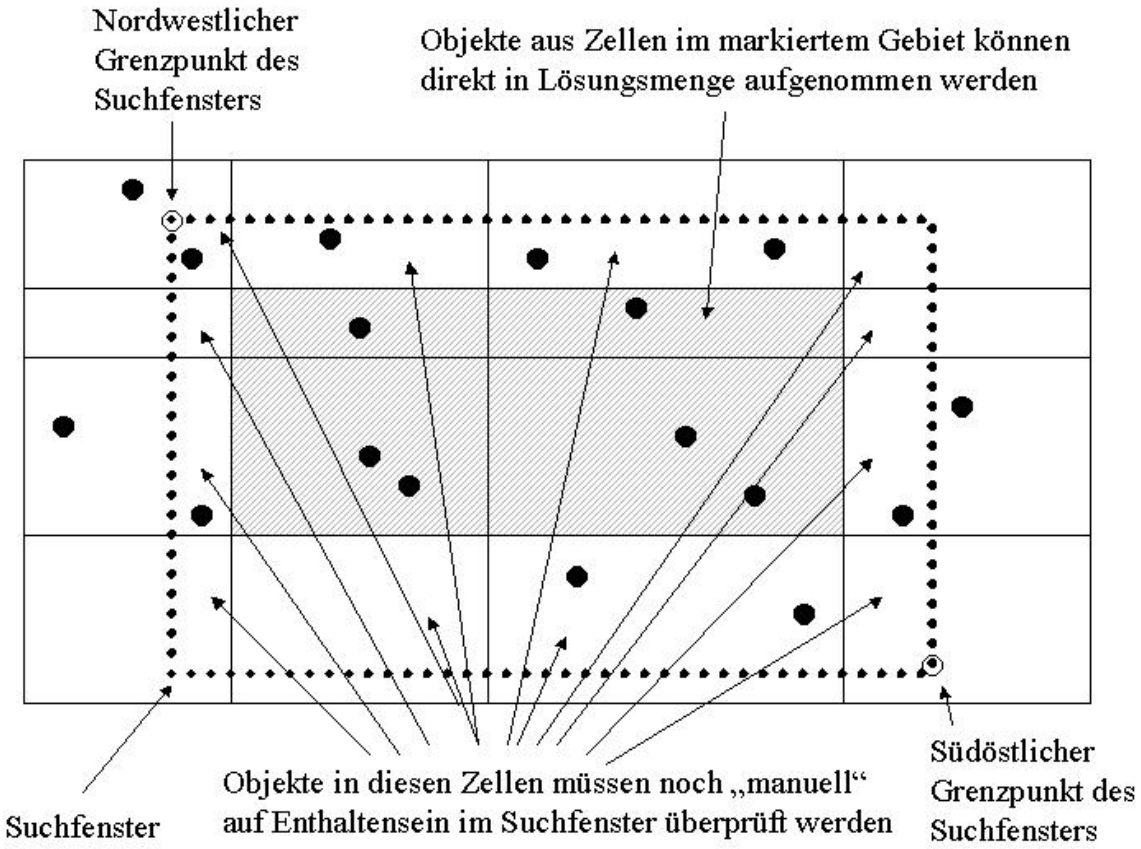


Abbildung 19: Vorgehen bei Bereichsanfrage in einem Grid-File

Die soeben beschriebene Eigenschaft läßt eine effektive Implementierung von Bereichsanfragen mit dem Grid-File zu. Vor allem kommen die Vorteile dabei für große Suchfenster zum Tragen, denn in diesem Fall werden besonders viele Enthaltenseinsüberprüfungen eingespart, da für große Suchfenster der „innere“ Bereich der überdeckten Zellen, deren Objekte direkt in die Lösungsmenge übernommen werden können, entsprechend größer ist, als für kleine Suchfenster. Für kleine Suchfenster ist es dagegen häufig der Fall, daß durch den Rand schon alle berührten Zellen abgedeckt sind, und die dort gehaltenen Objekte müssen dann, wie oben beschrieben, noch auf Enthaltensein im Suchfenster überprüft werden.

Eine typische, im Rahmen dieser Arbeit erstellte Implementierungsvariante für eine Nächster-Nachbar-Anfrage bedient sich zur Beantwortung der Frage nach dem nächstgelegenen Objekt zum Suchobjekt *s* mit oberer Distanzschranke *d* der folgenden Schritte.

- Finde die Zelle im Rasterverzeichnis, in der das Suchobjekt liegen würde, bzw. liegt.
- Berechne den nächsten Nachbar aus den in der Zelle gespeicherten Elementen.

- Finde alle Zellen, die noch Objekte von Interesse enthalten könnten. (Anschaulicher gesagt, werden diese Zellen im zweidimensionalen Fall vom Kreis um das Suchobjekt mit Radius *d* bestimmt. Alle Zellen, die von diesem Kreis geschnitten werden, werden markiert und im nächsten Schritt überprüft).
- Überprüfe alle markierten Zellen aus dem letzten Schritt, ob sich eines der in ihnen gehaltenen Objekte noch näher am Suchobjekt befindet, als das bisher gefundene vorläufige Ergebnis.
- Gib das Objekt zurück, das den kleinsten Abstand zum Suchobjekt hat.

Dieses Vorgehen unterscheidet sich von den bisher betrachteten Ansätzen für die „Nächster-Nachbar"-Anfrage derart, daß nicht rekursiv von der Wurzel her der Weg durch den Baum in Tiefentraversierung verfolgt wird, und auf diesem Weg alle in Frage kommenden Lösungskandidaten untersucht werden, sondern daß die Indexstruktur sozusagen von „unten nach oben" (engl.: Bottom to top) durchlaufen wird. Nach dem Feststellen der ersten Zelle werden nur noch die Zellen betrachtet, die potentielle Kandidaten enthalten können, dies kann durch Vergleichen des Mindestabstandes der Objekte einer Zelle mit dem Abstand des aktuell nächsten Nachbarn oder der in der Anfrage spezifizierten Maximalentfernung geschehen. Näheres hierzu kann z.B. in [HIN98] nachgelesen werden.

Da aber für das Grid-File durch die oben dargestellten Schwierigkeiten mit dem überlinearen Wachstum des Rasterverzeichnisses und dem Verzicht auf die Sekundärspeicherverwendung keine Garantien im Bezug auf die Kosten für den Zugriff auf ein Objekt gegeben werden können, wird die Leistungsfähigkeit von „Nächster-Nachbar"-Anfragen für die Indexstruktur Grid-File durch die in Kapitel 5 beschriebenen Tests experimentell überprüft.

SAF6: Günstige Speicherausnutzung

Zunächst soll wieder der mittlere Speicheraufwand pro gespeichertem Element O_{avg} untersucht werden. Dazu muß der Speicherverbrauch für die gesamte Struktur bekannt sein, dieser setzt sich aus dem Speicherverbrauch für das Halten des Rasterverzeichnisses und dem Verbrauch für das Halten der Zellen im Hauptspeicher zusammen.

Der Speicherverbrauch für das Halten des Rasterverzeichnisses setzt sich seinerseits für die im Rahmen dieser Diplomarbeit implementierte Version des Grid-Files zusammen aus den

- Kosten für das Halten eines Feldes von waagrechten Skalen.
- Kosten für das Halten eines Feldes von senkrechten Skalen.
- Kosten für das Halten eines Zeigers auf eine matrixartige Datenstruktur über die auf die einzelnen Zellen zugegriffen werden kann.
- Kosten für das Halten eines Zählers, der die Anzahl der gesetzten waagrechten Skalen beschreibt.
- Kosten für das Halten eines Zählers, der die Anzahl der gesetzten senkrechten Skalen beschreibt.
- Kosten für das Halten einer Konstante, die die maximal mögliche Anzahl von waag- und senkrechten Skalen und damit auch indirekt die maximale Anzahl von Zellen beschreibt, in die der zur Verfügung stehende Raum partitioniert werden kann.

Der Speicherverbrauch, den eine Zelle benötigt, setzt sich zusammen aus den

- Kosten für das Halten eines Zählers, der die Anzahl der momentan in dieser Zelle gehaltenen Objekte beschreibt,
- Kosten für das Halten eines Feldes von Verweisen, über das auf die einzelnen in dieser Zelle gehaltenen Objekte zugegriffen werden kann,
- Kosten für das Halten eines Wertes, der die maximale Anzahl von Objekten, die in einer Zelle gehalten werden können, beschreibt.

Damit ergibt sich der gesamten Speicherplatz, den die Indexstruktur Grid-File benötigt, aus folgender Formel:

- Speicheraufwand für das Rasterverzeichnis: O_{gd}
- Speicheraufwand für eine Zelle: O_c
- Anzahl der Zellen: n_c

$$O_{tot} = O_{gd} + O_c * n_c$$

Für den mittleren Speicheraufwand pro Element gilt demnach:

- Anzahl der Elemente: n_e

$$O_{avg} = \frac{O_{gd} + O_c * n_c}{n_e}$$

Die mittlere Hauptspeicherausnutzung U_{avg} ergibt sich laut Definition aus der Division der Größe des „sinnvoll“, d.h. für das Halten eines Objektes verwendeten Speichers durch die Größe des insgesamt durch diese Indexstruktur belegten Speichers.

Damit ergibt sich für U_{avg}:

- Durchschnittlich pro Zelle verwendeter Speicher: U_c
- Anzahl der Zellen: n_c

$$U_{avg} = \frac{U_c * n_c}{O_{tot}}$$

SAF7: Unterstützung von Sperrmechanismen
Auch für das Grid-File wurde bislang noch kein spezieller Mechanismus zur effizienten Synchronisierung paralleler Zugriffe vorgeschlagen. Analog zu den bisher betrachteten Indexstrukturen wurden daher Experimente mit dem „Write once/Read multiple“-Verfahren zur Synchronisierung durchgeführt. Die erzielten Ergebnisse werden dann in Kapitel 5.2.3 dargestellt.

4.5 Der R-Baum

Zuletzt soll die Indexstruktur R-Baum noch auf die Eignung als räumliche Datenhaltungskomponente für den Lokationsdienst der NEXUS-Plattform untersucht werden.

Die im Rahmen dieser Arbeit implementierte, vorgabenbedingt ausschließlich hauptspeicherbasierte Version des R-Baums besitzt nachfolgende erwähnenswerte Eigenschaften (siehe auch 2.4.4 für eine genauere Beschreibung der Indexstruktur R-Baum):

- Das Halten von punktförmigen Koordinatenangaben für die zu speichernden Objekte in den Blättern statt des Haltens von geometrischen Gebilden mit räumlicher Ausdehnung. Dies wurde zum einen eingeführt, um Vergleiche zwischen punktförmigen Suchobjekten und den Positionsangaben der gespeicherten Objekte mit weniger Leistungsverlust durchführen zu können. Der zweite Grund ist die bessere Vergleichbarkeit mit den anderen implementierten Indexstrukturen, die ja ebenfalls

definitionsgemäß nur Objekte mit punktförmiger Gestalt verwalten können. Klar ist, daß der R-Baum durch den Verzicht auf die Verwaltung ausgedehnter Objekte eine bessere Leistungsfähigkeit beim Ausführen von Operationen mit rein punktförmigen Objekten bietet, auf diese Weise aber auch den großen Vorteil der höheren Flexibilität verliert.

- Unterscheidung zwischen Blättern und Indizes. Anders als bei den bisher vorgestellten, baumartigen Indexstrukturen wird bei dieser Version eines R-Baums nicht nur funktional zwischen Index und Blatt unterschieden, sondern auch strukturell. Ein Blatt, das durch Einfügen eines Objektes und einem dadurch hervorgerufenen Überlauf in zwei neue Blätter aufgeteilt wird, wird nun auch strukturell als Index behandelt, indem dessen Datentyp dementsprechend umgewandelt wird. Diese auch strukturelle Unterscheidung wurde nötig, da in den Indizes Listen mit räumlich ausgedehnten geometrischen Gebilden, den MBBs, gehalten werden und in den Blättern stattdessen Listen mit punktförmigen Positionsangaben ohne räumliche Ausdehnung (siehe oben). Um den Speicheraufwand zu minimieren, wurde statt jedem Knoten unabhängig von der Funktion je eine Liste zum Halten der räumlich ausgedehnten Gebilde im Indexfall und eine Liste zum Halten der punktförmigen Positionsangaben im Blattfall beizufügen, die oben erwähnte Unterscheidung eingeführt. Eine zweite Unterscheidung, die zwischen Blatt und Index getroffen wird, besteht im Halten eines Feldes von Nachfolgerzeigern für Indizes, das bei einem Blatt ersetzt wird durch Felder zum Halten von Zeigern auf die Daten der gespeicherten Objekte. Auch hier wurde eine verbesserte Speicherausnutzung angestrebt.
- Es wurden zwei gebräuchliche, aber unterschiedliche Algorithmen implementiert, die bei einer Blattpartitionierung zur Verteilung der Objekte des geteilten Blattes auf die beiden neuen Blätter führen. Bei der ersten handelt es sich um einen Algorithmus, der in quadratischer Zeitkomplexität arbeitet. Dieser steckt relativ Aufwand in die Berechnung der besten Partitionierung. Der zweite Algorithmus arbeitet in linearer Zeitkomplexität und wählt dafür eine unter Umständen nicht optimale Aufteilung aus, was später bei Anfragen zum Verfolgen von mehr Pfaden führen kann, als bei der erstenVariante.

Als nächstes soll die Einhaltung der unter 2.4.1 vorgestellten allgemeinen Anforderungen für den R-Baum überprüft werden.

Zum ersten Punkt, *Erhaltung der topologischen Struktur*, ist festzustellen, daß sie beim R-Baum, wie bei vielen anderen mehrdimensionalen Indexstrukturen nur eingeschränkt erhalten bleibt. Zwei in der Realität nahe beieinander liegende Objekte werden nur dann im selben Blatt gespeichert, wenn diese bei jedem Schritt des Einsinkvorgangs in den Baum denselben Weg nehmen. Es kann also oft vorkommen, daß nahe beieinander liegende Objekte in unterschiedlichen Blättern gehalten werden; für den R-Baum existieren in jedem Fall keine speziellen Mechanismen, um dies auszuschließen.

Zum zweiten Punkt, *Anpassungsfähigkeit an variierende Objektdichten,* läßt sich sagen, daß der R-Baum, wie jede andere Datenstruktur, die nicht den gesamten zur Verfügung stehenden Raum überdeckt, sondern nur die Teile, die von Objekten (bzw. daraus resultierenden MBBs) besetzt sind, eine relativ gute Anpassungsfähigkeit bietet. Gebiete, in denen eine hohe Objektdichte vorherrscht, werden in mehr Teilgebiete partitioniert, Gebiete mit einer geringen Objektdichte werden dementsprechend seltener partitioniert.

Für den dritten Punkt, *Durchführung dynamischer Reorganisationsmaßnahmen*, kann dem R-Baum wieder ein gute Bewertung gegeben werden. So sind in der ursprünglich von Guttman vorgeschlagenen, und im Rahmen dieser Arbeit implementierten Version des R-Baums (und den meisten der auf ihm basierenden Vorschläge) beispielsweise Mechanismen zur Sicherstellung einer Mindestbelegung eines Blattes nach Löschen eines der in diesem Blatt gehaltenen Objekte enthalten. Diese Mechanismen arbeiten nicht rein lokal auf dem betroffenen Blatt. Stattdessen werden Änderungen, die weiter oben im Baum Auswirkungen haben können, nach oben propagiert. Ein Beispiel hierfür ist die Hochpropagierung des nach Löschen eines Objektes geschrumpften MBB dieses Blattes. Die von Guttman vorgeschlagenen Mechanismen zur Teilung eines Blattes in zwei neue Blätter und der nachfolgend durchgeführten Verteilung der im ursprünglichen Blatt gehaltenen Objekte auf die beiden neuen Blätter führt zu einer ungefähren Gleichbesetzung der beiden Blätter, um so neue Teilungen die bei einem erneuten Überlauf eines dieser Blätter geschehen müßten, weitgehend hinauszuzögern. Von allen hier betrachteten mehrdimensionalen Indexstrukturen ist der R-Baum die am besten balancierte und ausgeglichene und damit am besten dynamisch reorganisierende Indexstruktur.

SAF1: Stabilität bei dynamischem Zugriffsverhalten

Wie bereits gesagt, ist die Indexstruktur R-Baum die stabilste der hier betrachteten Strukturen. Sie ist die einzige Indexstruktur, die beispielsweise Garantien bezüglich der Höhe und der Besetzungsrate der Knoten abgibt. Die Höhe beträgt dabei meist $\lfloor log_m N \rfloor - 1$, wobei mit m die untere Schranke für gespeicherte Objekte in einem Blatt, bzw. Nachfolger in einem Index und mit N die Anzahl der in der Indexstruktur insgesamt zu speichernden Objekte bezeichnet werden. Die Besetzungsrate der Baumknoten mit Ausnahme der Wurzel beträgt im schlechtesten Fall $\frac{m}{M}$, wobei mit M die maximale Anzahl der in einem Knoten zu haltenden Objekte bzw. Nachfolger bezeichnet wird.

In die Aufrechthaltung der Balancierung und die Mindestbelegung von Knoten wird einiger Aufwand gesteckt, der Lohn dafür ist die Garantie des Nichtentartens, die ein R-Baum geben kann, d.h. es kann beispielsweise kein Entarten zur linearen Liste möglich sein.

Ein anderer positiver Aspekt, den der R-Baum bietet, ist dessen Unabhängigkeit von der Einfügereihenfolge. Ein R-Baum hat für eine Menge von einzufügenden Objekten immer denselben Aufbau, ein Objekt befindet sich immer im selben Blatt usw.. Festgelegt wird dieser Aufbau ganz entscheidend durch die Wahl der Partitionierungspolitik bei der Spaltung eines Blattes, wovon im nächsten Abschnitt noch die Rede sein wird.

SAF2: Effiziente Ausführung „schreibender" Zugriffe

Problematisch ist beim R-Baum dagegen das Leistungsverhalten von schreibenden Operationen (Einfügen, Löschen, Aktualisierung). Da wie schon häufig erwähnt, auf die Implementierung eines Sekundärspeicherverwaltungsmechanismus verzichtet wurde, kann der R-Baum die ihm eigenen Vorteile beim Aus- und Einlagern von Speicherseiten nicht nutzen. Dies verschlechtert das durch die oben angesprochenen Balancierungs- und Reorganisationsmaßnahmen ohnehin nicht optimale Leistungsverhalten für die schreibenden Operationen noch zusätzlich.

Einfügeoperationen werden bei einem R-Baum nach [GUT84] so ausgeführt, daß zunächst das passende Blatt durch Ausführen der Methode `chooseLeaf` ausgewählt wird. Das Auffinden eines solchen Blattes erfordert zum Teil aufwendige Berechnungen, je nach gewählter Partitionierungspolitik. Zum Beispiel muß bei gewählter Politik „Geringste Vergrößerung" (engl.: Least Enlargement Policy) die theoretische Vergrößerung, die der

MBB jedes Nachfolgers bei Aufnahme des einzufügenden Objektes mitmachen müßte, berechnet werden. Danach muß derjenige Pfad weiter verfolgt werden, der die geringste potentielle Vergrößerung besitzt. Eine andere gebräuchliche Politik ist auch „Geringste Überlappung“, hier wird der Zweig ausgewählt, deren durch Aufnahme des Objektes unter Umständen vergrößerte MBB die geringste Überlappung mit den MBBs der anderen Nachfolger besitzt. Diese Politik wird beispielsweise beim R*-Baum verwendet. Nach dem Finden eines geeigneten Blattes müssen nach dem Einfügevorgang eventuelle Änderungen nach oben propagiert und dort jeweils entsprechend eingetragen werden, dies kann nach [GUT84] Änderungen bis hin zur Wurzel mit sich führen.

Das Löschen eines Objektes aus einem R-Baum wird in ähnlicher Art und Weise durchgeführt. Zunächst wird durch Ausführen der Methode `findLeaf` das Blatt bestimmt, das das zu löschende Objekt enthält. Wiederum kann dies zum Teil zu einer ganzen Reihe von Berechnungen führen. Wird ein solches gefunden und enthält dieses das zu löschende Objekt, so wird der entsprechende Eintrag gelöscht und die Änderungen wieder nach oben propagiert (z.B. Verkleinerung der MBB dieses Blattes). Gleichzeitig wird, wie oben schon erwähnt, überprüft, ob das Blatt noch mehr oder gleich viel Objekte als gefordert enthält, und es unter Umständen mit einem anderen Blatt verschmolzen bzw. eine Neuverteilung der gehaltenen Objekte auf andere Blätter durchgeführt.

Wie sich erkennen läßt, leistet der R-Baum wesentlich mehr Aufwand, um nach bestimmten Heuristiken optimale Verteilungen von Objekten auf Knoten durchzuführen als die anderen bisher diskutierten Indexstrukturen. Dies führt zu einem merklichen schlechteren Leistungsverhalten bei allen schreibenden Operationen.

Aktualisierungsoperationen werden durch die Eigenschaft des R-Baums unterstützt, den MBB jedes Knotens im Baum zu halten. Statt nun also bei der Aktualisierung der Position eines Objektes dieses aus dem Baum zu löschen und danach wieder einzufügen, kann zunächst überprüft werden, ob sich bei simpler Änderung der Positionsdaten des Objektes der MBB des Blattes ändert. Ist dies der Fall, so muß das Objekt wie oben beschrieben gelöscht und neu eingefügt werden, im anderen Fall jedoch ist es ausreichend, eine lokale Änderung der Positionsinformation für das betreffende Objekt durchzuführen.

SAF3: Effizienz bei ungleichmäßigen Datenverteilungen

Der R-Baum ist als einzige der hier betrachteten Indexstrukturen gut für die Verwaltung von Objekten, die in einer ungleichmäßigen Verteilung vorliegen, geeignet. Starken Häufungen von Objekten in bestimmten Teilgebieten wird so begegnet, daß dann die Wahrscheinlichkeit groß ist, diese im selben Blatt zu speichern, was zu einem verbesserten Zeitverhalten bei Anfragen führt. Dies hat seinen Grund im relativ aufwendigen Vorgehen, mit dem das Einfügen eines Objektes durchgeführt wird. Hier wird versucht, die optimale Einfügeposition für ein Element, abhängig von dessen Position zu finden (siehe auch 2.4.4). Elemente, die nahe beisammen liegen, werden so mit großer Wahrscheinlichkeit im selben Teilbaum, bzw. Blatt gespeichert, da der MBB eines Knotens bei Hinzufügen eines neuen Elementes aus einer „Anhäufung" nur geringfügig vergrößert werden muß. Daraus resultiert die genannte Beobachtung. Erkauft wird die gute Leistungsfähigkeit für lesende Zugriffe allerdings durch einen stark erhöhten Aufwand bei den schreibenden Zugriffen (Erhaltung der balancierten Struktur bei Einfügen und Löschen eines Elementes, Finden von optimalen Verteilungen von Elementen auf Teilbäume).

SAF4: Einfachheit

Die Verwaltung des R-Baums ist durch die vielen Reorganistionsmaßnahmen weit schwerer zu durchschauen, als dies bei den anderen bisher besprochenen Baumstrukturen möglich ist. Die die Balancierung des R-Baums erhaltenden Methoden sind recht aufwendig und erfordern das Unterscheiden und Bearbeiten einer Reihe von Fällen. Daher läßt sich sagen, daß die balancierte, gleichmäßige Struktur des R-Baums durch einen großen Komplexitätszuwachs gegenüber den unbalancierten Baumstrukturen erkauft wird.

SAF5: Effiziente Ausübung der Operationen „Bereichsanfrage" und „Nächster-Nachbar-Anfrage"

Bei Bereichsanfragen wird im R-Baum ausgehend von der Wurzel rekursiv im Baum abgestiegen, wobei immer die Nachfolgerzeiger verfolgt werden, deren zugeordnete MBB gemeinsame Punkte mit dem Suchfenster besitzen. Überdeckt das Suchgebiet die MBB eines Knotens vollständig, so müssen alle Unterbäume weiter betrachtet werden. Gelangt die Suche zu einem Blatt, so werden die dort gehaltenen Objekte auf Enthaltensein im Suchfenster überprüft und bei positivem Ergebnis in die Lösungsmenge übernommen.

Anfragen vom Typ „Nächster-Nachbar“ werden bei einem R-Baum üblicherweise so implementiert, daß ausgehend von der Wurzel rekursiv in Tiefentraversierung im Baum abgestiegen wird. Ist der aktuell betrachtete Knoten ein Index, so wird die Suche über diejenige Nachfolgezeiger fortgesetzt, deren zugeordnete MBB den Suchpunkt enthalten. Im anderen Fall, wenn der Knoten ein Blatt ist, wird unter den dort gehaltenen Objekten die beste Lösung gesucht und als vorläufiges Ergebnis festgehalten. Wenn die Suche schließlich endet, wird das zu diesem Zeitpunkt festgehaltene vorläufige Ergebnis als endgültiges Ergebnis übernommen.

Es wird deutlich, daß die Leistungsfähigkeit der beiden oben beschriebenen Anfragen ganz entscheidend davon abhängt, wie schnell bestimmte Pfade durch den Baum ausgeschlossen werden können. Dazu ist die gewählte Heuristik beim Einfügevorgang von größter Bedeutung. Die vorher angesprochene Heuristik „Geringste Vergrößerung“ zielt darauf ab, durch möglichst kleine MBBs zu einer möglichst kleinen Überlappung dieser zu kommen, um so schnell bestimmte Pfade ausschließen zu können. Die andere Heuristik, „Geringste Überlappung“ setzt diese Überlegung direkt um, und wählt den Pfad, dessen vergrößerter MBB tatsächlich die geringste Überlappung mit den anderen MBBs dieses Knotens aufwies. Der Rechenaufwand ist für eine solche Überprüfung natürlich größer als für die erste, da hier zusätzlich zur Berechnung des Flächeninhalts (Volumens,...) noch das Schnittgebiet mit jedem der anderen MBBs dieses Knotens berechnet und verglichen werden muß.

Wie schon weiter oben angesprochen, wurden zwei unterschiedliche Algorithmen zur Auswahl des geeigneten Nachfolgers gemäß der „Geringste Vergrößerung“-Heuristik implementiert: Zum einen ein in quadratischer Laufzeit und zum anderen ein in linearer Laufzeit ausführbarer Algorithmus. Diese beiden Algorithmen unterscheiden sich in der Qualität der Lösung: Die vom ersten Algorithmus gelieferten Lösungen haben die wesentlich höhere Wahrscheinlichkeit, die bestmögliche zu sein, als die vom zweiten Algorithmus gelieferte. Die Garantie, immer die beste aller möglichen Lösungen zu liefern, gibt allerdings keiner der beiden Algorithmen. Hierfür existiert ein naiver Algorithmus, der diese Garantie bieten kann, allerdings von erschöpfender Natur ist: Für jede Überprüfung werden alle möglichen Ergebnisse berechnet und das jeweils beste als Lösung ausgewählt. Dies führt zur Überprüfung von ungefähr 2^{M-1} Möglichkeiten pro besuchtem Knoten (ein zu spaltender Knoten enthält M Einträge). Da für M der Wert 50 nicht unüblich ist, ist der erschöpfende Algorithmus nicht in der Praxis einsetzbar, weil dann der Leistungsverlust bei

Einfügevorgängen nicht mehr tolerierbar ist. Näheres zu diesem Thema ist wieder in [GUT84] zu finden.

In den durchgeführten Experimenten zeigte sich, wie stark die Leistungsfähigkeit von Anfragen von dem Aufwand abhängig ist, der in den Einfügevorgang gesteckt wurde, die Ergebnisse sind in Kapitel 5 dargestellt.

SAF6: Günstige Speicherausnutzung
Auch hier soll zunächst einmal der Speicherplatz, der für einen Knoten eines R-Baums nötig ist, angegeben werden. Da bei der im Rahmen dieser Arbeit implementierten Version des R-Baums eine nicht nur funktionale Unterscheidung zwischen Indizes und Blättern getroffen wurde, wird im folgenden der Speicherverbrauch für diese beiden Fälle gesondert besprochen.

Die Kosten für einen Index eines R-Baums setzen sich zusammen aus:

- Den Kosten für das Halten eines Felds von Nachfolgerzeigern.
- Den Kosten für das Halten eines Felds von MBBs für die Nachfolger. Wenn diese besetzt sind, so fallen noch die Kosten für das Halten dieser MBB zusätzlich an. Die Summe dieser zusätzlichen Kosten für alle Indizes wird folgend mit „zusätzliche Indexkosten“ bezeichnet:

    1. Kosten für das Halten des nordwestlichen Begrenzungspunktes,
    2. Kosten für das Halten des südöstlichen Begrenzungspunktes.

    Für jeden dieser beiden Punkte reichen zwei Angaben aus, um diese festzulegen: z.B. Geografische Länge und Breite (jeweils zweidimensionaler Fall).

Die Kosten für ein Blatt eines R-Baums setzen sich zusammen aus:

- Den Kosten für das Halten eines Felds von Zeigern auf räumliche Objekten.
- Den Kosten für das Halten eines Felds von eindeutigen Bezeichnern von räumlichen Objekten.

Zusätzlich verursacht jeder Knoten, unabhängig, ob dieser ein Index oder ein Blatt ist, die folgenden Kosten:

- Kosten für das Halten einer Höhenangabe innerhalb des Baums (die Wurzel hat immer die Höhe `null`).
- Kosten für das Halten eines Zeigers auf den Baum selbst (direkter Zugriff auf die Wurzel).
- Kosten für das Halten eines Zeigers auf den Vorgänger (um Änderungen nach oben propagieren zu können).
- Kosten für das Halten eines Zeigers auf die diesen Knoten beschreibenden MBB.
- Kosten für das Halten einer Angabe, wieviele Objekte momentan in diesem Knoten gehalten werden.

Damit ergibt sich für den Gesamtspeicheraufwand O_{tot}:

- Kosten für Index: C_i
- Anzahl Indizes: n_i
- Zusätzliche Kosten für Index gemäß oben: C_{ia}
- Kosten fürBlatt: C_l
- Anzahl Blätter: n_l
- Zusätzliche Kosten für Knoten gemäß oben: C_{na}
- Anzahl Knoten: n_n

$$O_{tot} = C_i * n_i + C_l * {}_{nl} + C_{an} * n_n + C_{ia}$$

Daraus läßt sich der mittlere Speicheraufwand pro Element O_{avg} bestimmen

- Anzahl der Elemente: ne

$$O_{avg} = \frac{C_i * n_i + C_l * n_l + C_{an} * n_n + C_{ia}}{n_e}$$

Die mittlere Hauptspeicherausnutzung U_{avg} ermittelt sich durch Division der Größe des „sinnvoll“ belegten Speichers durch die Größe des insgesamt belegten Speichers. Als

Definition für „sinnvoll“ genutzen Speicher soll auch hier das Belegen eines Zeigers mit einer Adresse ungleich des JAVA-Datentyps `null` verwendet werden, eine nicht „sinnvolle“ Verwendung wäre demnach ein Allokieren von Speicher, der niemals zum Halten eines Zeigers gebraucht wird, sondern immer den initialen Wert `null` besitzt.

Der insgesamt allokierte Speicher zum Halten der Verweise, O_{all}, läßt sich folgendermaßen berechnen:

- Maximale Anzahl von Objekten pro Knoten: n_{max}
- Kosten für einen Verweis: C_p
- Anzahl der Knoten: n_n

$$O_{avg} = \frac{C_i * n_i + C_l * n_l + C_{an} * n_n + C_{ia}}{n_e}$$

Die Addition von 1 zu n_{max} erklärt sich so, daß jedes Feld aus Gründen der einfacheren Partitionierung im Fall des Überlaufs ein Element mehr aufnehmen kann, als dies die maximale Anzahl von Objekten pro Knoten angibt.

Zur Berechnung des „sinnvoll“ genutzen Speichers muß für jeden Knoten überprüft werden, zu welchem Grad dieser besetzt ist, d.h. welchen Wert die zugeordnete Variable zur Angabe der momentanen Besetzungszahl hat.

Der „sinnvoll“ genutzte Speicherplatz O_u läßt sich dann folgendermaßen berechnen:

- Anzahl der besetzten Plätze in Knoten Nummer i: n_i

$$O_u = 2 * C_p * \sum_{1}^{nn} (n_i + 1)$$

Damit läßt sich die mittlere Hauptspeicherausnutzung U_{avg} berechnen als:

$$U_{avg} = \frac{O_u}{O_{all}} = \frac{\sum_{1}^{nn} n_i + 1}{(n_{\max} + 1) * n_n}$$

SAF7: Unterstützung von Sperrmechanismen

Anders als bei den anderen, bisher betrachteten Indexstrukturen, wurden für den R-Baum im Laufe der Jahre mehrere Vorschläge für eine effiziente Synchronisierung paralleler Zugriffe veröffentlicht.

Banks, Kornacker und Stonebraker [BAN95] schlagen vor, zur Synchronisierung eine an den B-Link-Baum [LEH81] angelehnte, um Zeiger auf Knoten derselben Ebene erweiterte R-Baumstruktur, den sogenannten R-Link-Baum zu verwenden. Dem R-Link-Baum wird über die Geschwisterzeiger und eindeutigen Knotenbezeichner eine Totalordnung für die Knoten aufgeprägt. Diese wird verwendet, um später Situationen konkurrierender Art, die beim parallelen Zugriff vorkommen können, aufzulösen. Ein weiterer Vorteil, den diese Methode mit sich bringt, ist, daß über die eingeführte Totalordnung der Aufbau des Baums zu jedem Zeitpunkt feststellbar ist und abrufbar wird. Dies ist insbesondere günstig, wenn ein Verfahren zum Wiederanlaufen nach einem Absturz o.ä. nötig werden würde, da der Baum dann einfach durch Auslesen der entsprechenden Informationen rekonstruierbar wäre.

Ein anderer Vorschlag wurde von Kanth, Serena und Singh [KAN97] gemacht, der zu einer effiziente Synchronisierung paralleler Zugriffe während Modifizierungen der MBB von Indizes, etwa bei Einfügeoperationen, schon wieder parallele Zugriffe auf die Indexstruktur zuläßt. Weiterhin wird dieses Modell noch derart erweitert, daß dann auch während Partitionierungsvorgängen in Knoten noch Zugriffe auf die Datenstruktur durchgeführt werden können. Möglich wird dies unter anderem durch lokale Kopien von Knoten, auf denen Operationen zuerst durchgeführt werden und deren Änderungen erst später im Baum bekannt gemacht werden. Die Verfasser behaupten, mit ihrer Methode einen bis zu zweimal höheren Durchsatz als bisher zu erzielen. Testumgebung war dabei die Indexierung von digitalem Bildmaterial auf einer Mehrprozessormaschine.

Aus Aufwandsgründen wurde allerdings auf die Implementierung eines dieser Vorschläge verzichtet. Ohne den späteren Ergebnissen der angestellten Experimente vorzugreifen, wurden die Leistungswerte des R-Baums schon beim Zugriff nur eines Klienten im Vergleich mit den anderen untersuchten Indexstrukturen als zu schlecht angesehen, um diesen als Datenhaltungskomponente des Lokationsdienstes zu verwenden.

5 Experimente

In diesem Kapitel sollen alle mit den implementierten räumlichen Indexstrukturen durchgeführten experimentellen Untersuchungen ausführlich dargestellt werden. Hierfür wird zunächst in Unterkapitel 5.1 die verwendete Testumgebung und die Gründe für deren so geartete Beschaffenheit vorgestellt werden. Anschließend werden im folgenden Unterkapitel die Ergebnisse, die sich für die einzelnen untersuchten Indexstrukturen mit der entwickelten Testumgebung ergaben, dargestellt.

Um die zweite Entwicklungsalternative, das Basieren der Datenhaltungskomponente auf einer herkömmlichen Datenbank abzudecken, wurde im Laufe dieser Arbeit neben der entworfenen hauptspeicherbasierten auch eine Lösung entwickelt, die auf einer räumlichen Erweiterung für die bekannte *IBM DB2 Universal Database*, dem sogenannten *Spatial Extender*, basiert. Im letzten Unterkapitel wird deshalb zunächst eine kurze Einführung in den *Spatial Extender* gegeben, bevor auch hier die erzielten Ergebnisse dargestellt werden.

5.1 Testumgebung

Die Testumgebung, mit der die untersuchten Indexstrukturen auf ihre Eignung als Datenhaltungskomponente des Lokationsdienstes überprüft werden sollten, mußte in der Lage sein, das Verhalten einer Indexstruktur in verschiedenen Anwendungsfällen aufzuzeigen. Deshalb wurden mehrere verschiedene Testklassen entworfen und implementiert, die dieser Vorgabe Rechnung tragen sollten.

Die erste und einfachste Testklasse, die für alle implementierten räumlichen Indexstrukturen entwickelt wurde, ist dazu geeignet, die Leistungsfähigkeit einer Indexstruktur für das Durchführen der Operationen „Einfügen eines Objektes“, „Löschen eines Objektes“, „Ausführen einer Positionsanfrage“, „Ausführen einer Bereichsanfrage mit kleinem Suchbereich“, „Ausführen einer Bereichsanfrage mit mittlerem Suchbereich“, „Ausführen einer Bereichsanfrage mit großem Suchbereich“, sowie „Ausführen einer Nächster-Nachbar-Anfrage“ vergleichbar zu machen (`kdTreeTest`, `QuadTreeTest`,...). Dazu wurde je eine Klasse pro untersuchter Indexstruktur entworfen, in der durch Setzen der entsprechenden

Variablen eine gewisse Anzahl von Einfüge- und Löschvorgängen, sowie Positions-, Bereichs- und Nächster-Nachbar-Anfragen blockweise in Stapelverarbeitung durchgeführt wird. Für jeden dieser Blöcke gleichartiger Operationen wird dann die verstrichene Zeit von Beginn des Blockes bis Ende des Blockes festgehalten. Da durch diesen Test vor allem das grundlegende Zeitverhalten für die jeweiligen Operationen herausgefunden werden sollte, wurde hier auf die Verwendung mehrerer paralleler Threads, die zum Teil konkurrierend auf die Indexstruktur zugreifen, verzichtet. Stattdessen wurde nur ein einziger Thread aufgesetzt, der die einzelnen Operationen sequentiell abarbeitete. Um auch Experimente mit ungleich verteilten Datensätzen durchführen zu können, wurde in dieser Testklasse die Möglichkeit implementiert, bewußt Datensätze mit einer solchen Verteilung erzeugen zu lassen. Mit Hilfe einer zu setzenden boolschen Variable kann zwischen den beiden Alternativen „Erzeuge Datensätze in Gleichverteilung“ und „Erzeuge Datensätze in ungleichmäßiger Verteilung“ gewählt werden.

Der Zweck der folgend implementierten Testklasse ist es, die Leistungsfähigkeit einer Indexstruktur bei parallelen Zugriffen von mehreren Klienten zu überprüfen. Zur Simulation mehrerer Klienten wurde pro Klient je ein separater Thread aufgesetzt.

Diese Testklasse, `ConcurrentTest`, nutzt die erste Testklasse, indem hier eine festzusetzende Anzahl von Threads der ersten Testklasse nahezu gleichzeitig gestartet wird, die dann zum Teil konkurrierend auf die gemeinsam benutzte Indexstruktur zugreifen. Jeder dieser Threads arbeitet die festgelegten Operationsblöcke stapelartig ab, konkurrierende Zugriffe werden dabei durch die für die jeweilige Indexstruktur implementierten Sperrmechanismen synchronisiert. Auf die Bestimmung der Bearbeitungsdauer für die einzelnen Operationsblöcke der verschiedenen Threads wurde in dieser Testklasse verzichtet, das Hauptaugenmerk wurde hier auf die Feststellung der Effizienz des jeweiligen Synchronisierungsverfahrens gelegt. Auch in diesem Fall wurde als Kriterium der Effizienz die Größe der Zeitspanne ausgewählt, die zwischen dem Beginn der ersten Operation aller Threads und dem Ende der letzten Operation aller Threads lag.

Die dritte und letzte Testklasse, `LongTimeTest`, wurde entworfen, um auch Testläufe über längere Zeiträume unter dynamischen Bedingungen durchführen zu können Dazu wurde wieder nur ein Thread aufgesetzt, da es prinzipiell für das Verhalten einer Indexstruktur egal ist, ob Einfüge- oder Löschoperationen von einem Thread, oder von mehreren Threads

ausgeführt werden. Im Gegensatz zu den beiden oben beschriebenen Testklassen wird in dieser lediglich die Gesamtanzahl von durchzuführenden Operationen vor Beginn eines Testlaufs durch Setzen einer Variable festgelegt. Danach wird zufällig eine der folgenden Operationen ausgewählt und abgearbeitet:

- Einfügen eines Objektes mit einer zufällig gewählten Position und einer zufälligen eindeutigen Kennung.
- Löschen eines zufällig ausgewählten Objekts aus der Menge der zuvor eingefügten Objekte.
- Aktualisieren der Position eines gespeicherten Objektes auf eine neue, zufällig ausgewählte Position.
- Ausführen einer Bereichsanfrage mit Suchbereich mittlerer Größe und fester Position.
- Ausführen einer Nächster-Nachbar-Anfrage mit festem Suchradius und festem Suchzentrum.

Um eine Grundlast von Objekten bereitzustellen, wird vor dem eigentlichen Beginn der zufälligen Operationenabfolge initial eine gewisse, ebenfalls festlegbare Anzahl von Objekten eingefügt. Um die Testergebnisse für die verschiedenen Indexstrukturen vergleichbar zu machen, wird außerdem laufzeitbegleitend überprüft, ob sich die Anzahl der in der Struktur gespeicherten Objekte stets zwischen zwei festzulegenden Schranken befindet. Operationen, die zum Unter- bzw. Überschreiten einer solchen Schranke führen würden, werden nicht ausgeführt, sondern stattdessen eine andere Operation zufällig ausgewählt.

Die hier zuletzt geschilderte Testklasse ist vor allem zur Erforschung der Aspekte der „günstigen Speicherausnutzung“ gedacht. Für diesen Zweck wurden für jede der Indexstrukturen Methoden entwickelt, die entsprechende Informationen zurückliefern. Auf diese Weise soll für eine Indexstruktur deren Verhalten gemäß den in Kapitel 4 beschriebenen Metriken für langen Einsatzzeiträume unter dynamischen Bedingungen erforscht werden.

Zur Repräsentation von Positionsinformationen wurden die im Rahmen des NEXUS-Projektes entwickelte Klasse `GeodeticCoordinate`, die die Funktionalität geografischer Koordinaten umsetzt, verwendet. Räumlich ausgedehnte Gebiete, wie Suchfenster o.ä.,

wurden durch die Klasse `Segment`, die zur Darstellung achsenparalleler zweidimensionaler Gebiete entworfen wurde, dargestellt.

Bei der für die Durchführung der Experimente verwendeten Hardware handelte es sich um einen Rechner mit Intel Pentium III Prozessor und 128 MB Hauptspeicher, als Betriebssystem wurde Microsoft Windows 98 verwendet. Dies führt zu folgenden für die Speicherausnutzung relevanten Werten:

- Alle Zeiger benötigen 4 Byte (hardwareabhängig),
- das Halten einer Fließkommazahl benötigt 8 Byte (JAVA-Datentyp: `double`),
- das Halten eines ganzzahligen Wertes benötigt 4 Byte (JAVA-Datentyp: `int`).

5.2 Experimente mit räumlichen Indexstrukturen

Im folgenden sollen die Ergebnisse der im letzten Unterkapitel vorgestellten Testklassen dargestellt und bewertet werden. Eine Diskussion der Ergebnisse wird dann in Kapitel 6 angestellt. Dieses Unterkapitel gliedert sich in drei Teile, wobei in jedem Teil das Verhalten jeder der besprochenen Indexstrukturen für einen der drei oben vorgestellten Testfälle dargestellt wird.

5.2.1 Einzeltest

In diesem Unterkapitel werden nachfolgend die Ergebnisse, die sich für die erste Testklasse zur Untersuchung der Leistungsfähigkeit der einzelnen Indexstrukturen für schreibende Vorgänge und einfache Anfragen ergaben, dargestellt.

Für jede der zu untersuchenden Indexstrukturen wurde dabei die erste Testklasse folgendermaßen parametrisiert:

- Einfügen von 100000 normalverteilten räumlichen Objekten, wobei es sich bei diesen um punktförmige Objekte handelte.
- Ausführen von 50000 Positionsanfragen.
- Ausführen von je 10000 Bereichsanfragen mit einem Suchfenster der Abmessungen 100 x 100 Meter, 1000 x 1000 Meter und 10000 x 10000 Meter.

- Ausführen von 50000 Nächster-Nachbar-Anfragen.
- Löschen von 10000 Objekten.

Für jeden dieser Blöcke wurde die benötigte Bearbeitungszeit gemessen, die so gefundenen Ergebnisse werden in nachfolgenden Diagrammen dargestellt.

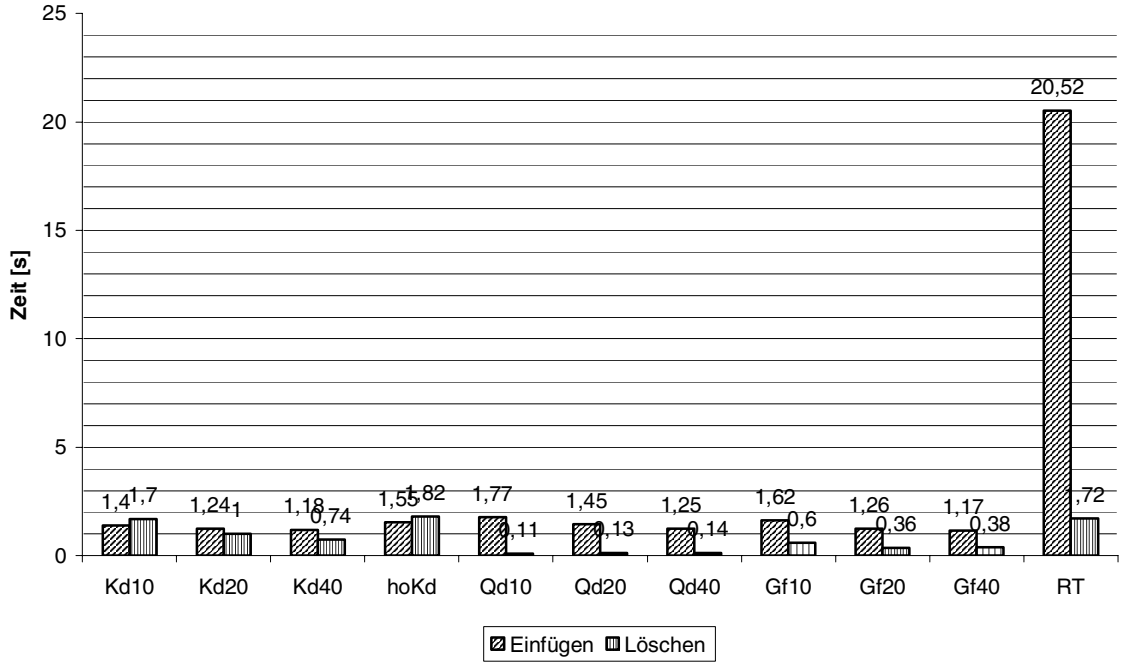
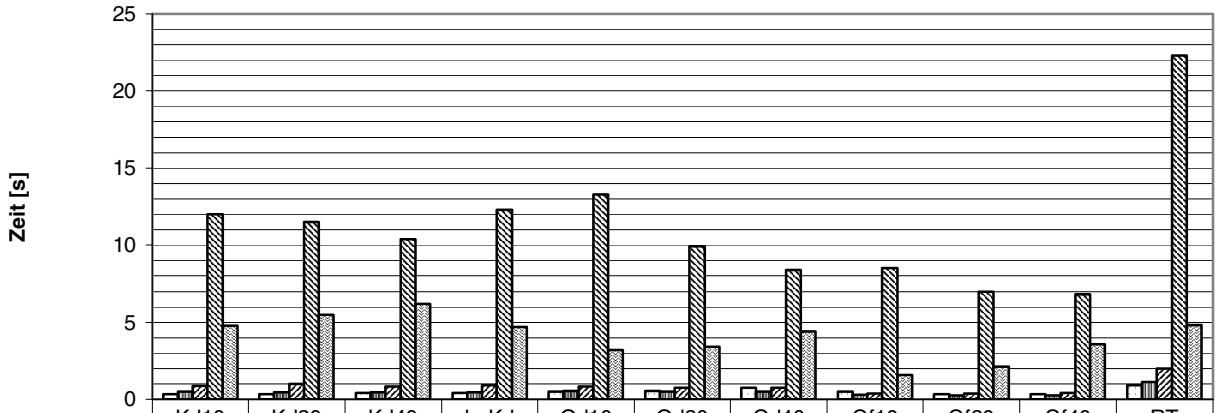


Abbildung 20: : Performance schreibender Operationen



	Kd10	Kd20	Kd40	hoKd	Qd10	Qd20	Qd40	Gf10	Gf20	Gf40	RT
PosAnf	0,35	0,35	0,44	0,43	0,49	0,55	0,75	0,51	0,35	0,33	0,92
BerAnf1	0,5	0,45	0,45	0,48	0,55	0,52	0,52	0,32	0,26	0,28	1,12
BerAnf2	0,89	1	0,82	0,91	0,86	0,77	0,74	0,4	0,39	0,44	2
BerAnf3	12	11,5	10,4	12,3	13,3	9,92	8,4	8,5	7	6,81	22,3
NNAnf	4,8	5,5	6,2	4,7	3,2	3,4	4,4	1,6	2,14	3,59	4,83

Abbildung 21: Performance lesender Operationen

Aus den oben stehenden Diagrammen ist ersichtlich, daß die Leistungsfähigkeit des R-Baums für die gewählte Testumgebung weit hinter denen der anderen Indexstrukturen zurückbleibt. Dies läßt sich nur zum Teil durch den großen Aufwand erklären, den der R-Baum in das Finden der optimalen Einfügeposition für ein Objekt steckt, obwohl dieser Punkt ein sehr gewichtiger ist: Mit Hilfe eines *Profilierers*, einem Werkzeug, mit dem u. a. eine Aufschlüsselung der insgesamt gebrauchten Laufzeit erreicht werden kann, wurde festgestellt, daß ein großer Laufzeitanteil durch das Erzeugen oder Anpassen der MBB des R-Baums hervorgerufen wurde. Dies mag eine JAVA-proprietäre Beobachtung sein, zukünftige Forschungen könnten dieses Verhalten auch für andere Implementierungssprachen untersuchen.
Ein anderer Grund ist, daß die gewählte Testumgebung für einen R-Baum nicht sonderlich gut geeignet war: Erstens wurden nur punktförmige Objekte verwendet und die Möglichkeiten des R-Baums, auch räumlich ausgedehnte Objekte zu verwalten, also nicht ausgenutzt. Damit wurden die Indexstrukturen, die von vorneherein nur zum Verwalten von punktförmigen Objekten entworfen wurden, bevorzugt. Zweitens wurde der R-Baum mit dem Gedanken im Hinterkopf entworfen, eine Sekundärspeicherverwaltung anbieten zu können. Da auf diese Möglichkeit bekanntlich verzichtet wurde, wurden so wiederum die rein hauptspeicherbasierten Indexstrukturen Quad-Tree und kd-Baum bevorzugt. Drittens ist zu bemerken, daß nur beim R-Baum Mechanismen zur dynamischen Reorganisation implementiert wurden, während bei den anderen Strukturen bislang keine derartigen Mechanismen berücksichtigt wurden. Denkbar wäre hier z.B. ein Verfahren zur Zusammenlegung schlecht ausgelasteter, benachbarten Knoten o.ä..

Die Leistungsfähigkeit der anderen implementierten Indexstrukturen war durchweg zufriedenstellend. Sehr überraschend war in diesem Zusammenhang die Leistungsfähigkeit des Grid-Files, daß sich in den meisten Bereichen gegen die rein hauptspeicherbasierten Baumstrukturen behaupten konnte. Vor allem seien hier die für die beiden Anfragetypen „Bereichsanfrage“ und „Nächster-Nachbar-Anfrage“ erzielten Ergebnisse erwähnt, in denen das Grid-File besonders überzeugte. Bei den Positionsanfragen waren die beiden Baumstrukturem dem Grid-File dagegen wieder überlegen. Dies könnte aber seinen Grund darin haben, daß die Skalen der implementierten Version des Grid-Files als einfaches sortiertes Feld gespeichert werden, daß bei jedem Suchvorgang sequentiell von vorne durchlaufen wird. Denkbar ist hier eine Verwaltung der Skalen beispielsweise mit Hilfe eines B-Baums, um so zu einer besseren Leistung zu kommen.

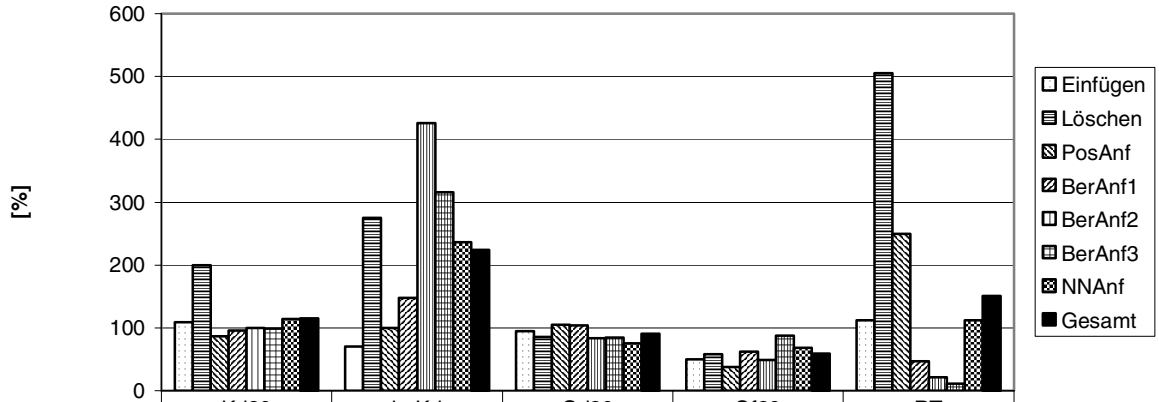
Interessant ist bei allen untersuchten Indexstrukturen, mit Ausnahme des R-Baums und des homogenen kd-Baums, der ja nur ein Objekt pro Knoten aufnehmen kann, die starke Abhängigkeit der Leistungsfähigkeit der Anfragen von der Aufnahmefähigkeit der Zellen, bzw. Baumknoten. Generell läßt sich hier sagen, daß sich durch eine Erhöhung der maximal aufnehmbaren Objekte einer Zelle/Knotens eine Verschlechterung der Leistung von „Bereichsanfragen“, aber eine Verbesserung der Leistung von „Nächster-Nachbar-Anfragen“ ergibt. Dies läßt sich so erklären, daß bei einer Bereichsanfrage nach Überprüfung, ob ein Knotens/Zelle komplett vom Suchfenster überdeckt ist, alle dort gehaltenen Objekte ohne weitere Überprüfung in die Lösungsmenge übernommen werden können und nur für die nur teilweise überdeckten Knoten/Zellen eine Überprüfung der einzelnen Objekte auf Enthaltensein im Suchfenster nötig ist. Durch Vergrößerung der Zell/Blattkapazität werden also mehr Objekte ohne Prüfung in die Lösungsmenge übernommen als zuvor. Dies drückt sich im sinkenden Zeitverbrauch einer solchen Anfrage aus. Für Nächster-Nachbar-Anfragen dagegen ist für jedes gefundene Objekt eine Überprüfung nötig, ob es näher am Suchpunkt liegt, als das bisher gefundene beste Ergebnis. Bei Vergrößerung der Zell/Blattkapazität werden damit pro durchsuchter Zelle mehr von diesen unter Umständen teuren Vergleichen (bei geografischen Koordinaten) nötig.

Weiterhin ist die sehr gute Leistung des Quad-Trees bei Löschvorgängen zu bemerken, im Gegensatz zu der des kd-Baums. Wie bereits in den letzten Kapiteln gesagt wurde, muß der kd-Baum nach dem Löschen bzw. Ändern eines Referenzobjektes alle im Baum darunter befindlichen Objekte neu einfügen, dies schlägt sich deutlich in den entprechenden Zeitwerten nieder.

Zuletzt soll hier noch auf die Überlegenheit des heterogenen über den homogenen kd-Baum angesprochen werden, die sich durch bessere Leistungen in nahezu allen Bereichen wiederspiegelt, unabhängig von dem maximalen Kapazität der Blätter des Ersteren.

Um das Verhalten der Indexstrukturen bei ungleichmäßig verteilten Datensätzen bewerten zu können, wurde die obige Testumgebung dann derart parametrisiert, daß nun statt dem Einfügen von 100000 gleichverteilten Objekten 100000 Objekte eingefügt wurden, die in einer solchen Verteilung vorlagen. Danach wurden wieder dieselben Operationen wie oben auf der jeweiligen Indexstruktur ausgeführt. Die Ausführungszeiten der einzelnen Blocks

wurden gemessen und ins Verhältnis zu den entsprechenden Zeitwerten gesetzt, die sich für die einzelnen Blocks für gleichverteilte Daten ergaben. Die relativen Änderungen zwischen den Zeitwerten der einzelnen Blocks sind in nachfolgendem Diagramm ablesbar. Zusätzlich wird hier auch der Durchschnittswert aller Veränderungen angegeben.



	Kd20	hoKd	Qd20	Gf20	RT
Einfügen	109	70	95	50	112
Löschen	200	275	85	58	505
PosAnf	86	100	105	37	250
BerAnf1	96	148	104	62	47
BerAnf2	100	426	83	49	21
BerAnf3	99	316	84	87	11
NNAnf	114	236	75	68	112
Gesamt	115	224	90	59	151

Abbildung 22: Prozentuale Performanceänderung bei Wechsel von gleichverteilten Datensätzen zu ungleichmäßig verteilten Datensätzen

Es ist einfach ablesbar, daß das Grid-File am besten mit den ungleichmäßig verteilten Datensätzen zurecht kam, was sich in den geringsten Steigerungsraten für die einzelnen Operationsblöcke ausdrückt. Dies läßt auch hier wieder den Schluß zu, daß sich mit dem Grid-File für kurze Einsatzzeiträume die besten Ergebnisse erzielen lassen. Erst bei länger dauernden Anwendungen läßt die Leistungsfähigkeit dieser Indexstruktur deutlich nach, so daß sich die in Kapitel 4 angegebene theoretische Betrachtung bestätigt.

Bemerkenswert ist auch der große Unterschied, den der R-Baum in den Steigerungsraten für die schreibenden und lesenden Zugriffe erkennen läßt. So ist die Steigerung der Ausführungszeit für die schreibenden Zugriffe deutlich höher als die der lesenden Zugriffe. Dies ist leicht verständlich, denn die guten Zeitwerte für Anfragen werden mit einem deutlich erhöhten Aufwand, um z.B. den optimalen Einfügeort für ein Element zu finden, erkauft.

5.2.2 Parallele Zugriffe

Hier sollen die Ergebnisse dargestellt werden, die sich bei Anwendung der zweiten entwickelten Testklasse zum Überprüfen des Verhaltens einer Indexstruktur bei parallelen, unter Umständen konkurrierenden Zugriffen von mehreren Klienten ergaben. Jede der betrachteten Indexstrukturen wurde dabei in zwei unterschiedlichen Versionen getestet. Bei der ersten Version werden alle Operationen gleich behandelt, d.h. es existiert keine Möglichkeit, lesende Operationen parallel ausführen zu können. Die zweite Version besitzt das in Kapitel 4 vorgestellte „Write once/Read multiple“-Synchronisierungsverfahren, das den parallelen Zugriff lesender Operationen ermöglicht, den gleichzeitigen schreibenden Zugriff, bei dem Inkonsistenzen entstehen können, aber ausschließt.

Es wurde entschieden, insgesamt 10 Threads aufzusetzen, um damit den parallelen Zugriff auf die Datenstruktur von 10 Klienten zu simulieren. Jeder dieser Threads mußte das folgende Operationspaket bearbeiten:

- Einfügen von 10000 normalverteilten Objekten.
- Löschen von 100 Objekten.
- Ausführen von je 1000 Bereichsanfragen mit einem Suchfenster der Abmessungen 100 x 100 Meter, 1000 x 1000 Meter und 10000 x 10000 Meter.
- Ausführen von 5000 Nächster-Nachbar-Anfragen.

Zur Einschätzung der erzielten Ergebnisse muß aber vorausschickend nochmals darauf hingewiesen werden, daß nur ein physischer Prozessor bei den Experimenten verwendet wurde, also keine echte Parallelität erzielt werden konnte. Die Testergebnisse können demnach bei Verwendung mehrerer Prozessoren von den hier gefundenen abweichen.

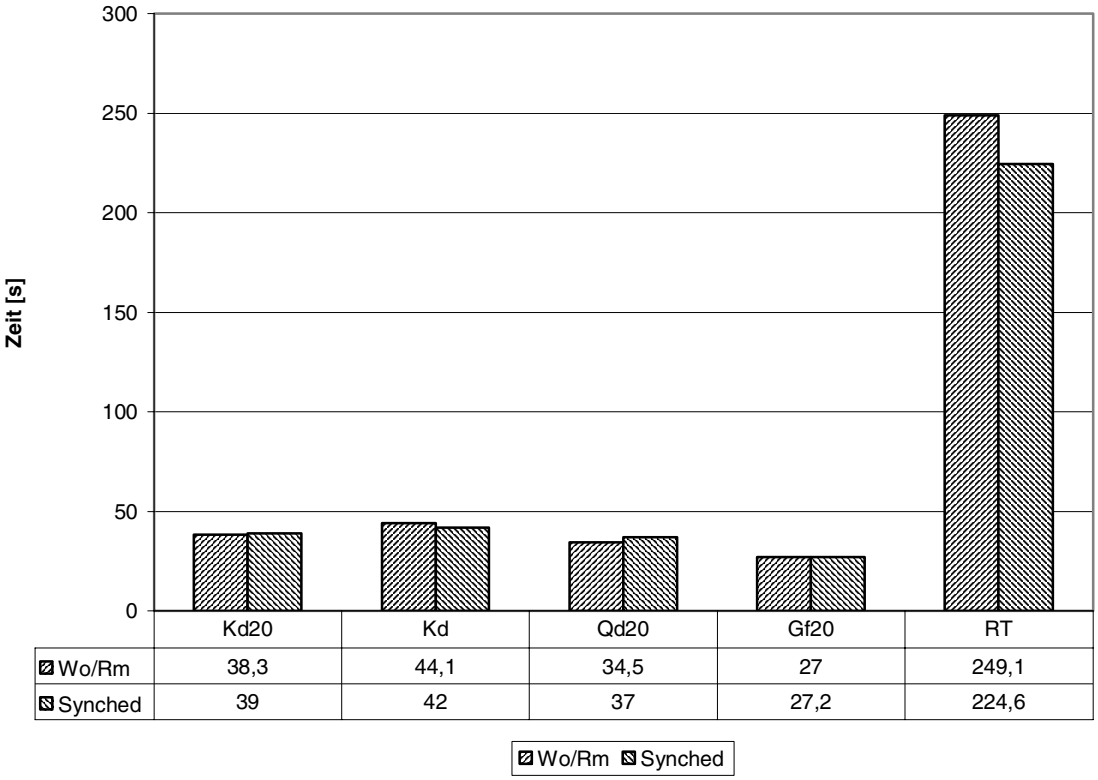


Abbildung 23: Vergleich der Synchronisationsverfahren "Voll-Synchronisierung" und "Write once/Read multiple"

Hier ist besonders augenfällig, daß das verwendete Synchronisierungsverfahren nicht für alle Indexstrukturen die gleiche Verbesserung bewirken. Während „Write once/Read multiple" für den Quad-Tree eine durchschnittliche Verbesserung von ca. 7 Prozent für das oben angegebene Operationspaket erbrachte, konnten beim heterogenen kd-Baum und beim Grid-File keine deutlichen Leistungsverbesserungen erzielt werden. Beim homogenen kd-Baum und beim R-Baum war sogar eine Verschlechterung zum vollsynchronisierten Fall feststellbar.

Eine mögliche Erklärung könnte sein, daß in JAVA die Synchronisierung mittels des `synchronized`-Schlüsselworts optimiert ist, und eventuell gewisse konkurrierende Zugriffe, sofern diese nicht zu inkonsitenten Zuständen führen würden, für den Benutzer transparent parallel durchgeführt werden können. So könnte der von „Write once/Read multiple" erzeugte zusätzliche Bearbeitungsaufwand für manche Anwendungsfälle eine schlechtere Leistung bewirken.

5.2.3 Langzeittest

Mit Hilfe dieser dritten Testklasse sollte das Verhalten der Indexstrukturen über einen längeren Zeitraum überprüft werden. Der Zugriff erfolgte einfacherweise nur über einen

Thread, da der Fokus hier auf der Untersuchung der Speicherausnutzung lag. Es wurde entschieden, die Testparameter folgendermaßen zu besetzen:

- Anzahl der initialen Einfügungen von normalverteilten Objekten: 10000,
- Obere Schranke von Objekten: 2000,
- Untere Schranke von Objekten: 500,
- Gesamtanzahl der auszuführenden Operationen: 2000000, 5000000, bzw. 20000000.

Dies führte zu drei unterschiedlich intensiven Testfällen, die erzielten Ergebnisse sind für alle drei Testfälle und alle Indexstrukturen bis auf den R-Baum, der ja bekanntermaßen als einzige der untersuchten Indexstrukturen Garantien bezüglich der Besetzung seiner Knoten und damit auch der Speicherausnutzung gibt, nachfolgend dargestellt. Die Speicherausnutzung usw. des R-Baums hängt damit nicht von der Dauer ab, die dieser dynamischen Einfüge- und Löschvorgängen ausgesetzt war, sondern bleibt annähernd konstant. Aus diesem Grund wurde auch in den entsprechenden Diagrammen auf das Darstellen der unterschiedlichen Lastpakete verzichtet, sondern nur ein Wert angegeben.

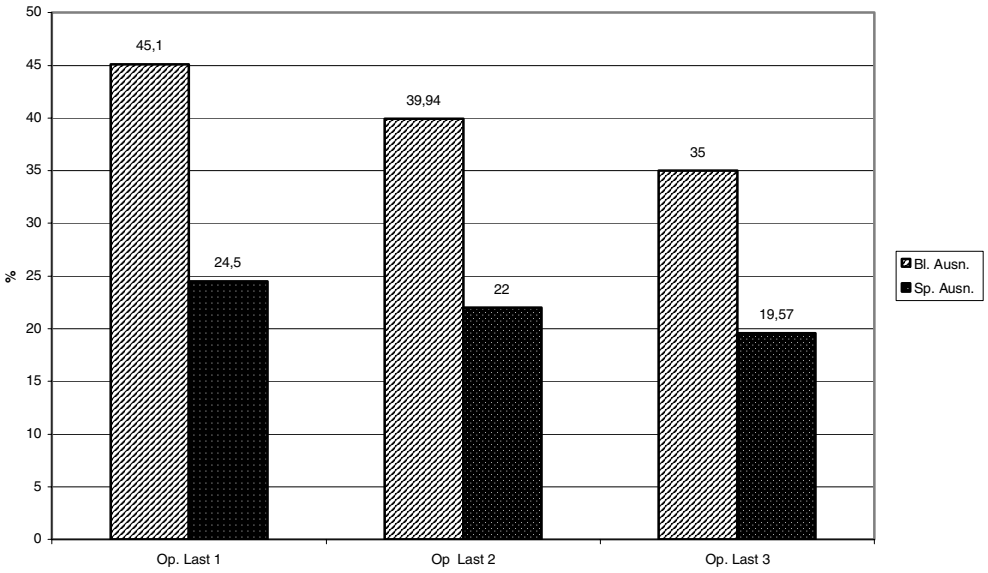


Abbildung 24: Blatt- und Speicherausnutzung des kd-Baums für steigend intensive Operationslasten

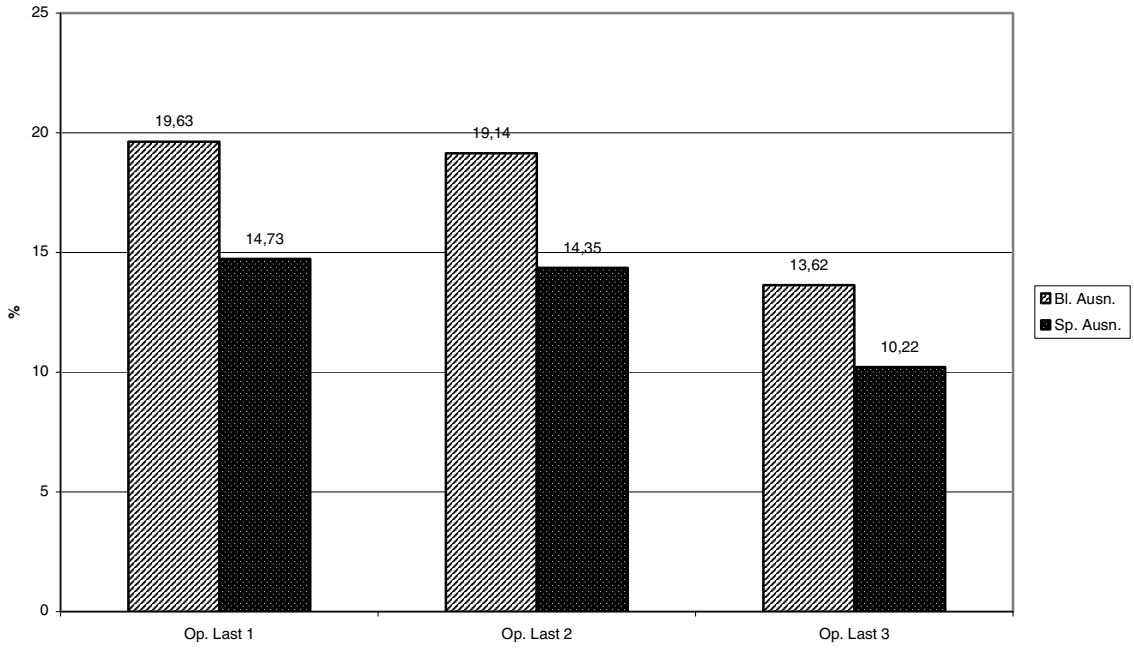


Abbildung 25: Blatt- und Speicherausnutzung des Quad-Trees für steigend intensive Operationslasten

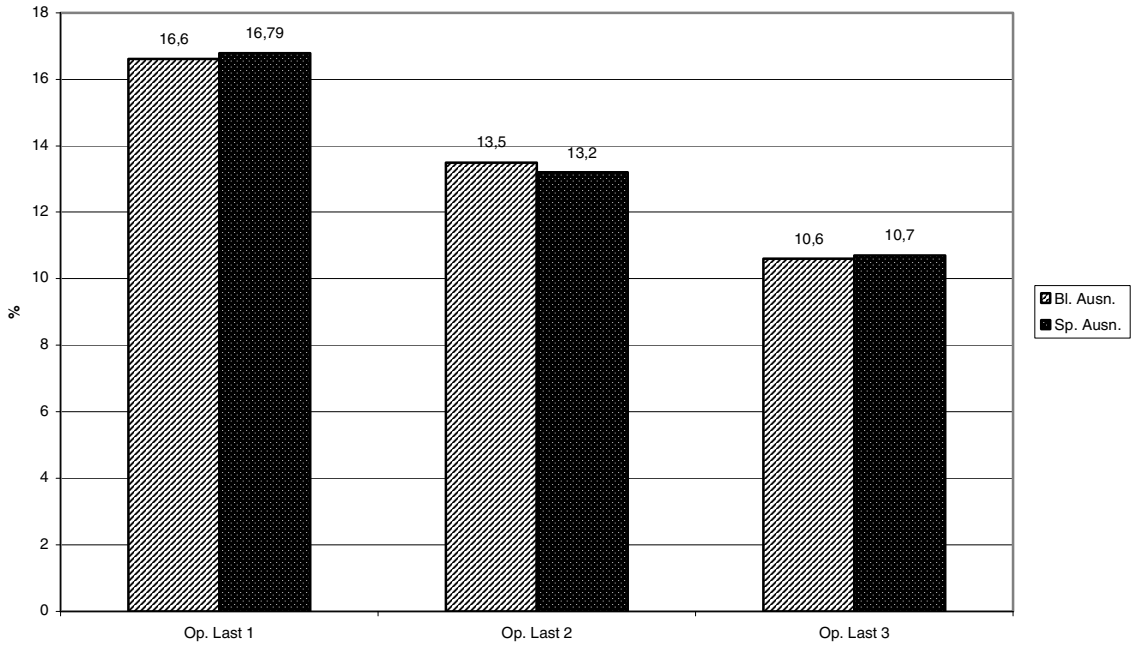


Abbildung 26: Blatt- und Speicherausnutzung des Grid-Files für steigend intensive Operationslasten

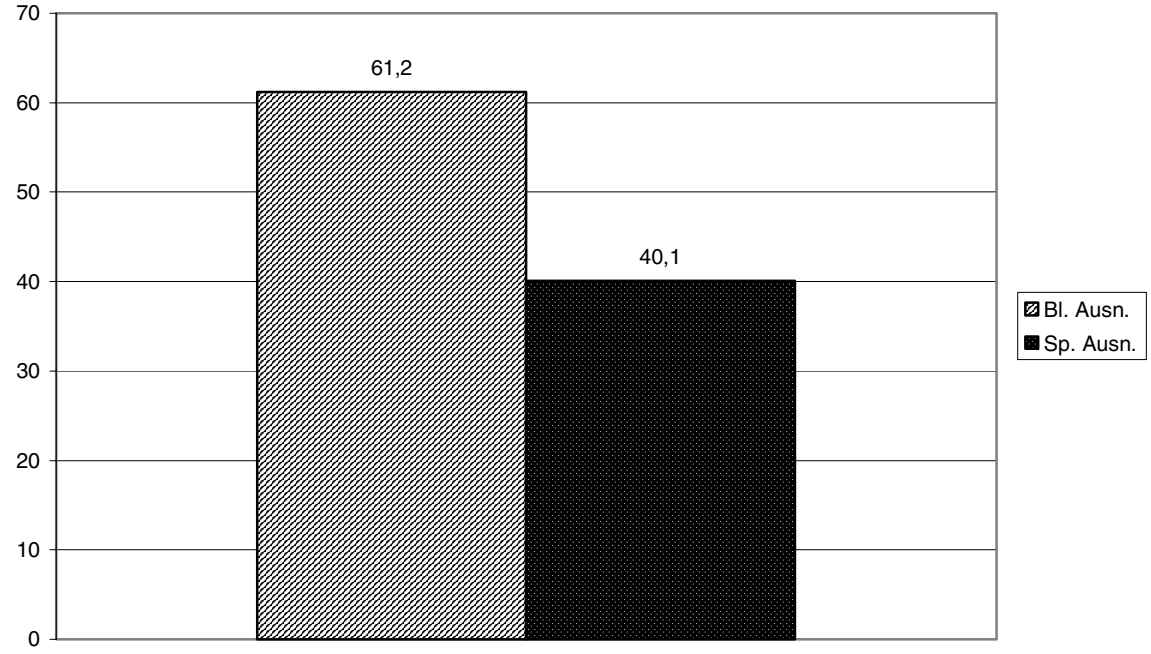


Abbildung 27: Blatt- und Speicherausnutzung des R-Baums für alle Operationslasten

Zunächst soll auf die Entwicklung der beiden Metriken „Speicherausnutzung“ und „Ausnutzung der Blätter/Zellen“ für die drei unterschiedlich intensiven Operationslasten eingegangen werden. Es zeigt sich an den obenstehenden Diagrammen, daß alle drei Indexstrukturen eine mit steigender Operationslast sinkende Blatt/Zellenausnutzung und Speicherausnutzung besitzen. Dies ist zurückzuführen auf die wachsende Unbalanciertheit der Bäume bzw. das überlinear wachsende Rasterverzeichnis des Grid-Files, sowie den weiter oben angesprochenen vorläufigen Verzicht auf dynamische Reorganisationsmethoden bei all diesen Strukturen. Deutlich ist der Zusammenhang zwischen mittlerer Blatt/Zellenausnutzung und mittlerer Speicherausnutzung zu erkennen: Viele schlecht besetzte Zellen bzw. Blätter führen unweigerlich zu einer schlechten Speicherausnutzung, da ja konstant viel Speicherplatz zum Speichern von Objekten bei Erzeugung eines Blattes oder einer Zelle belegt wird. Bei Vergrößerung der maximalen Kapazität eines Blattes bzw. einer Zelle werden sich diese Werte noch weiter verschlechtern, da ja nun noch mehr Speicherplatz pro Blatt/Zelle allokiert wird. Dies ist aber nicht in oberen Diagrammen erfaßt.

Die Diagramme zeigen weiterhin, daß das Grid-File mit steigender Operationslast die meisten Prozentpunkte in den beiden Ausnutzungsmetriken verliert, nämlich durchschnittlich ca. 35 Prozent, während der Quad-Tree ca. 23 Prozent, der kd-Baum gar nur 20 Prozent einbüßt. Als Erklärung dafür ist wieder das überlineare Wachstum des Rasterverzeichnisses und die große Anzahl von schlecht besetzten Zellen zu nennen, die ja bei Teilungsvorgängen einer Zelle entstehen. Den kleinen Vorsprung, den der kd-Baum vor dem Quad-Tree innehat, erklärt sich

dagegen aus dem Halten von mindestens einem Objekt pro Knoten beim kd-Baum, während der Quad-Tree Objekte ja nur in seinen Blättern hält und damit innere Baumknoten „verschwendeter“ Speicherplatz hinsichtlich dieser Metrik sind. Dies ist aber nicht allein der Grund für die letzte Beobachtung, die anhand der Diagramme erkennbar ist, nämlich der deutlich besseren Werte, die der kd-Baum hinsichtlich der beiden Ausnutzungen unabhängig von der vorangegangenen Operationslast besitzt. Eine mögliche Erklärung hierfür ist das Vorgehen bei Überlauf und Teilung eines Blattes. Beim kd-Baum wird ein Blatt in zwei neue Blätter geteilt, und alle Objekte bis auf eines (neues Referenzelement) auf diese verteilt, d.h. jedes der neuen Blätter enthält dann ca. $\frac{n_{max}}{2}$ Objekte (n_{max} = maximale Kapazität eines Blattes). Beim Quad-Tree wird bei Überlauf eines Blattes dieses aber in vier neue Blätter geteilt und die Objekte wieder auf diese verteilt, d.h. jedes dieser Blätter enthält nur noch ca. $\frac{n_{max}}{4}$ Objekte.

Bessere Werte werden, wie nicht überraschend ist, mit dem R-Baum erzielt: Dies allerdings unter anderem auf Kosten der Leistungsfähigkeit der angebotenen Operationen, wie dies im letzten Abschnitt ja besprochen wurde.

Der homogene kd-Baum liefert für diese Betrachtung das beste Ergebnis, da hier ja nur ein Objekt pro Knoten gehalten wird und damit immer eine hundertprozentige Blatt- und daraus resultierende Speicherausnutzung erzielt wird.

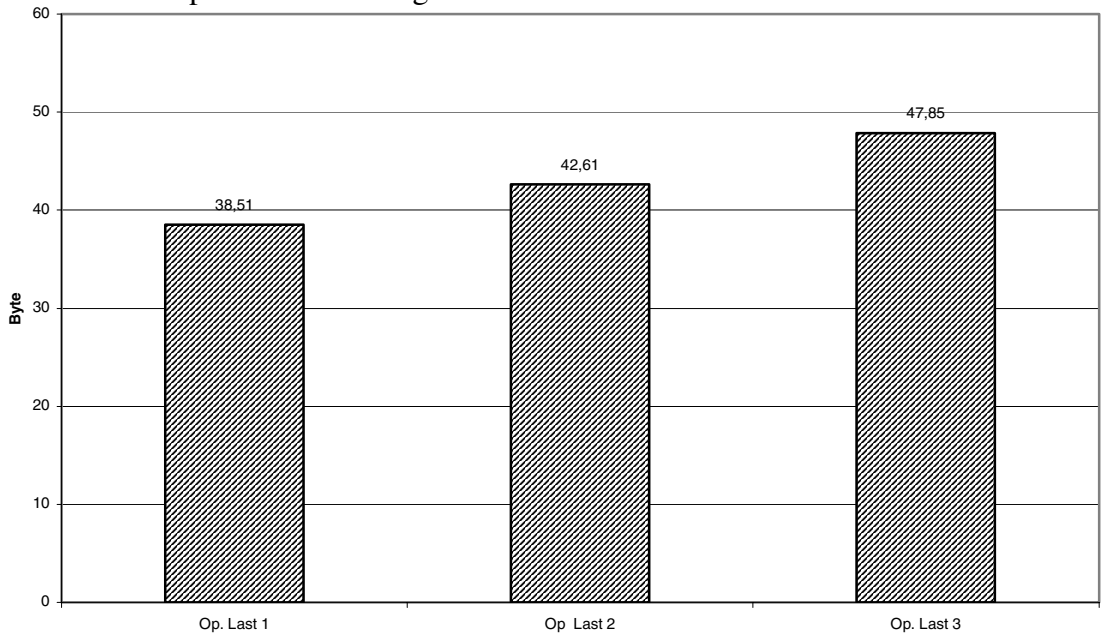


Abbildung 28: Mittlerer Speicheroverhead pro Element für den kd-Baum bei steigend intensiven Operationslasten

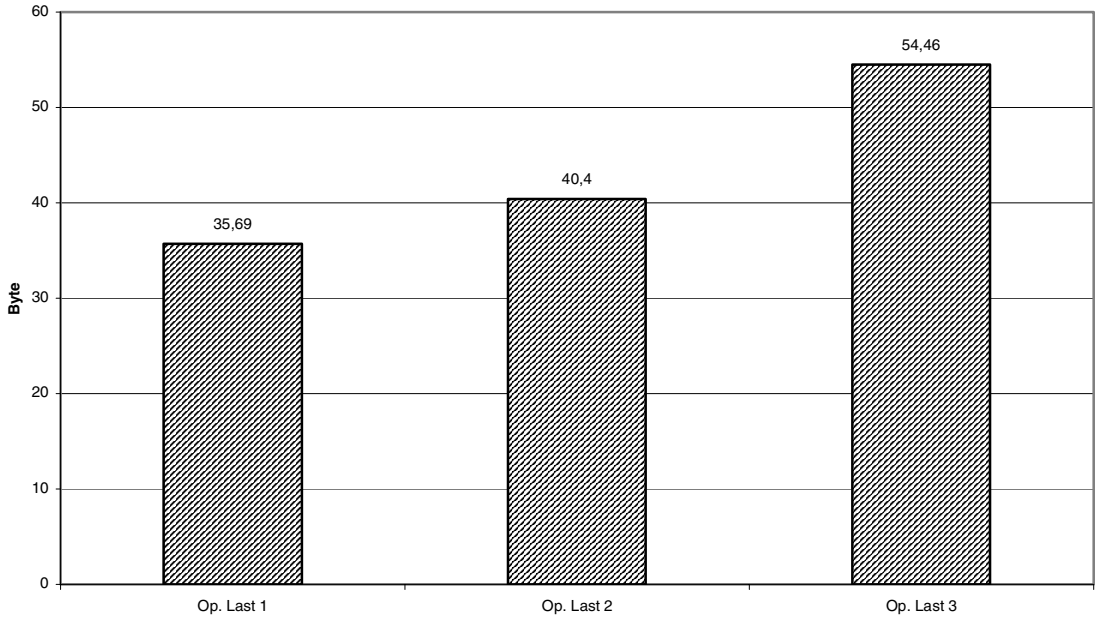


Abbildung 29: Mittlerer Speicheroverhead pro Element für den Quad-Tree bei steigend intensiven Operationslasten

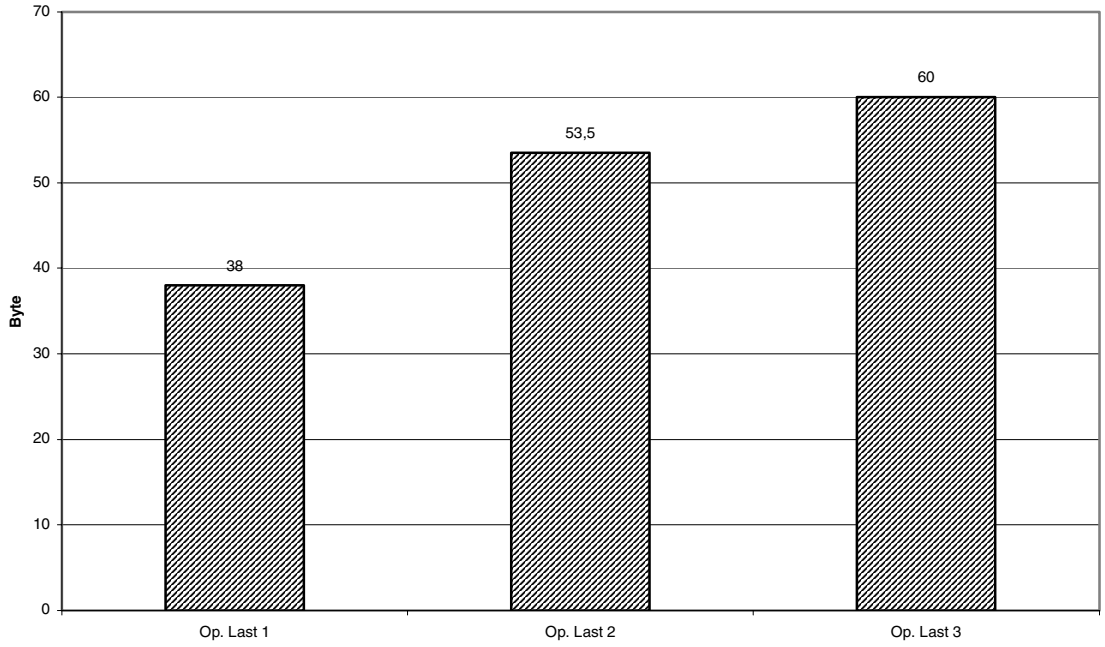


Abbildung 30: Mittlerer Speicheroverhead pro Element für den Quad-Tree bei steigend intensiven Operationslasten

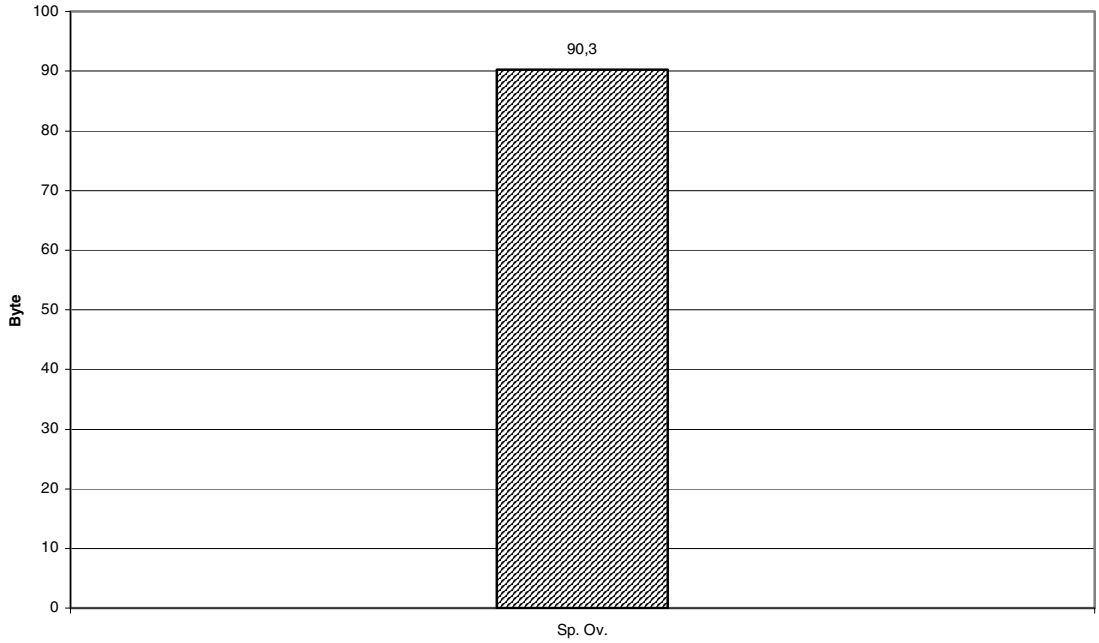


Abbildung 31: Mittlerer Speicheroverhead pro Element für den R-Baum bei allen Operationslasten

Als nächstes sollen die Diagramme für den mittleren Speicheraufwand pro gespeichertem Objekt für jede Indexstruktur kurz erläutert werden. Auch hier liefert der kd-Baum von allen drei Indexstrukturen zusammengefaßt die besten Ergebnisse. Dies gilt im besonderen für den homogenen kd-Baum, der nur einen konstanten Betrag von 24 Bytes benötigt. Deutlich zeigt sich auch, wie stark das Grid-File vom überlinear wachsenden Rasterverzeichnisses beinflußt wird. Dies ist im Anstieg des mittleren Speicheraufwands um ganze 100 Prozent nach Ausführen der größten Operationslast im Gegensatz zum Wert, der sich nach Ausführen der kleinsten Operationslast ergab, erkennbar. Beim Quad-Tree steigt dieser Wert um ca. 50 Prozent, beim heterogenen kd-Baum sogar nur um 25 Prozent an. Dies läßt sich mit denselben Argumenten wie schon bei der mittleren Speicherausnutzung erklären.
Für die kleinste Operationslast liefert das Grid-File allerdings das beste absolute Ergebnis aller drei Indexstrukturen. Der R-Baum, der ja einen nahezu konstanten Speicheraufwand bietet, liefert hier von allen betrachteten Indexstrukturen die schlechtesten Ergebnisse ab, dies hat seinen Grund im Halten der MBBs für jeden Knoten, die relativ viel Platz benötigen.

Diesen Abschnitt abschließen soll eine Untersuchung der Entartungsneigung der einzelnen Indexstrukturen. Dazu wurde einfacherweise nach dem Abarbeiten je eines Viertels eines Operationspaketes die dafür benötigte Zeitspanne gemessen und diese auf die Laufzeit des ganzen Paketes normiert. Die so erhaltenen Werte wurden dann in die oben dargestellten Diagramme eingetragen, um so herausfinden zu können, inwiefern die Leistungsfähigkeit einer bestimmten Indexstruktur unter einer möglichen Entartung leidet.

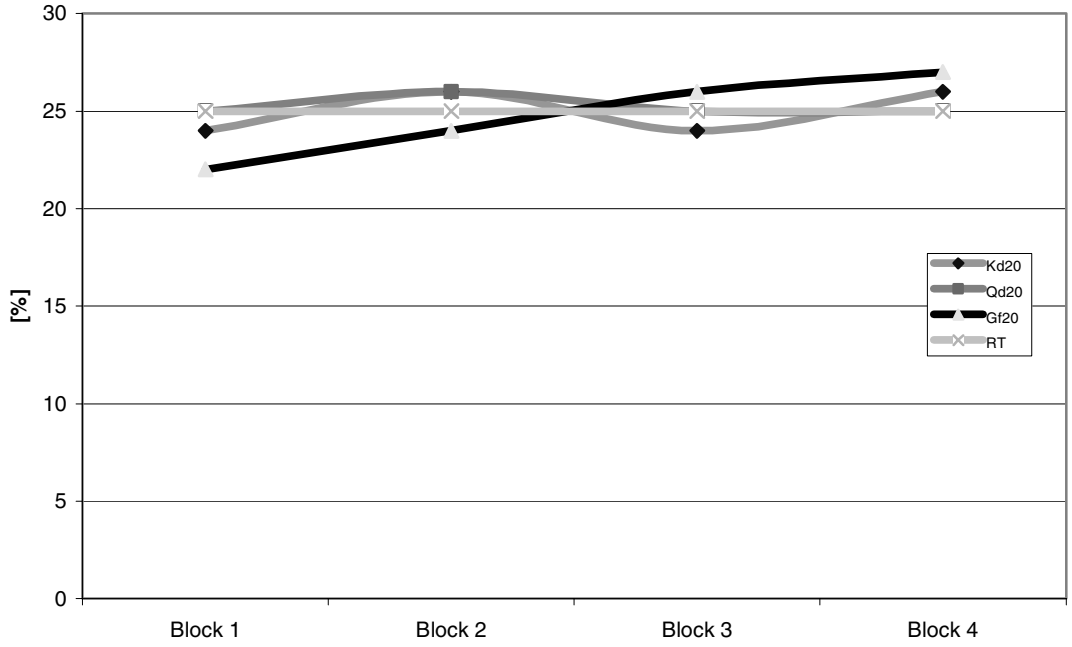


Abbildung 32: Zeitvergleich der zur Abarbeitung von je einem Viertel des gesamten Operationspaketes benötigten Zeit für normalverteilte Datensätze (2000000 Operationen)

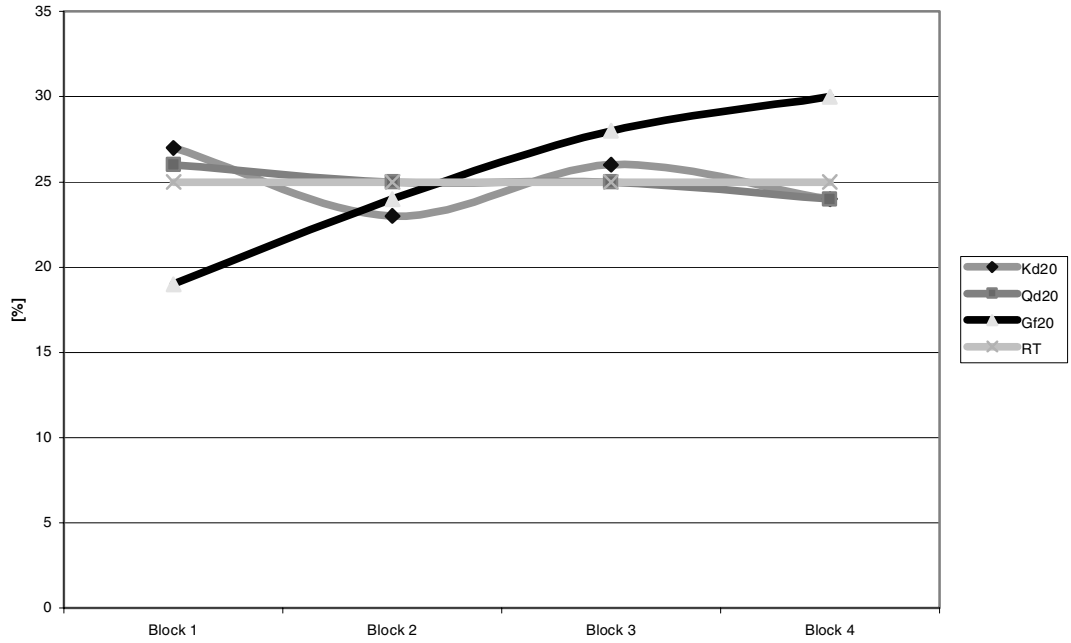


Abbildung 33: Zeitvergleich der zur Abarbeitung von je einem Viertel des gesamten Operationspaketes benötigten Zeit für normalverteilte Datensätze (20000000 Operationen)

An den obenstehenden Diagrammen ist deutlich ablesbar, daß das Grid-File hier wieder die schlechtesten Ergebnisse liefert, erkennbar an der für die beiden Operationslasten streng monoton steigenden Funktionskurve. Besonders deutlich wird dies bei der zweiten, größeren Operationslast, bei der der Anstieg am steilsten ausfällt. Dies läßt tatsächlich auf das Entarten (überlinear wachsendes Rasterverzeichnis und Anzahl von Skalen) dieser Indexstruktur zurückschließen. Der kd-Baum und der Quad-Tree lassen keine klare Tendenz erkennen, dies

läßt auf ein relativ unproblematisches Verhalten bei gleichverteilten Daten und dynamischem Einfüge- und Löschverhalten schließen. Beim R-Baum schwanken die einzelnen Werte kaum, offensichtlich, weil dieser zu jedem Zeitpunkt vollständig balanciert ist.

Zuletzt soll hier noch eine kurze Bewertung des Zeitverhaltens angestellt werden, die sich bei Einfügen von ungleichmäßig verteilten Objekten ergab. Dazu wurden für den dritten Testfall (20000000 Operationen) nur Objekte, die in einer solchen Verteilung vorlagen, eingefügt. Es wurde dann entsprechend zu oben der Anteil an der insgesamt benötigten Zeit für jedes der vier Viertel berechnet. Auch hier läßt sich nur für das Grid-File ein stetig ansteigender Anteil mit fortschreitender Einsatzdauer erkennen, während der kd-Baum und der Quad-Tree keine klare Tendenz bemerken lassen. Der Anstieg beim Grid-File ist allerdings nicht stärker als für den normalverteilten Fall, so daß man folgern kann, daß das Auftreten von ungleichmäßigen Verteilungen der Datensätze keine extrem kritische Verschlechterung der Leistungsfähigkeit für alle Indexstrukturen mit sich bringt. In dem untenstehenden Diagramm ist dieser Sachverhalt dargestellt.

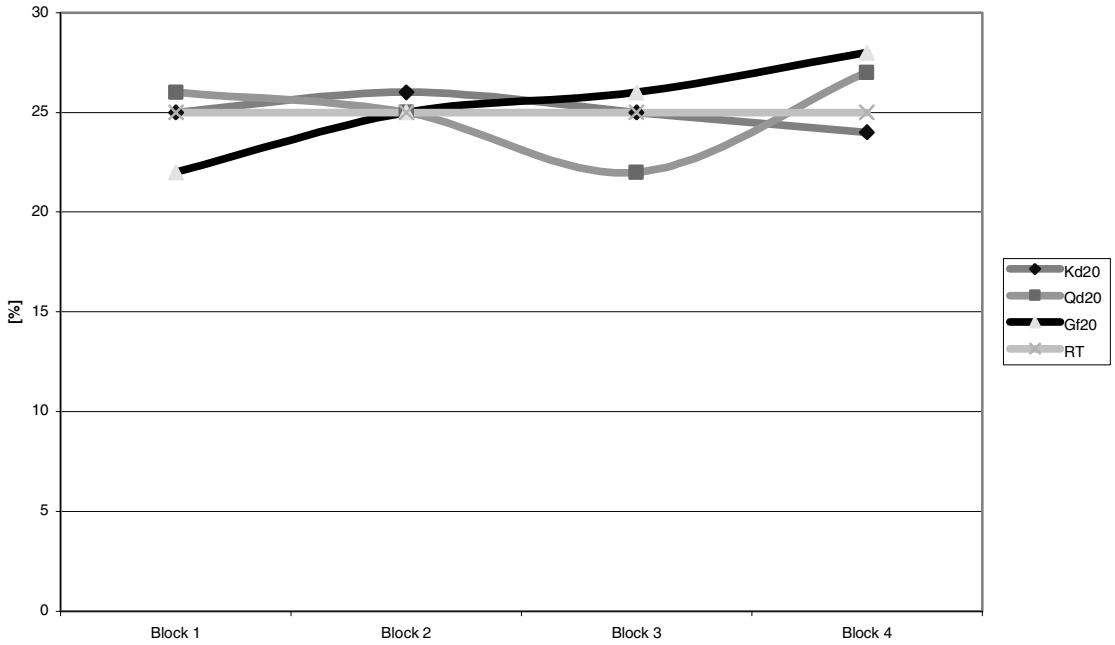


Abbildung 34: Zeitvergleich der zur Abarbeitung von je einem Viertel des gesamten Operationspaketes benötigten Zeit für ungleichmäßig Datensätze (20000000 Operationen)

5.3 Experimente mit DB2 Spatial Extender

Wie schon weiter oben erwähnt, wurde, um der zweiten Entwicklungsalternative Rechnung zu tragen, auch Untersuchungen mit herkömmlichen Datenbanken angestellt. Wie in Kapitel 2 dargestellt, reicht dabei für das effiziente Verwalten von räumlichen Daten und der schnellen Beantwortung von Anfragen ein herkömmliches relationales Datenbanksystem, wie etwa *IBM DB2 UDB* nicht aus. Stattdessen existieren für die meisten bekannten Datenbanksysteme kommerzielle Module, die auf ein solches herkömmliches Datenbanksystem aufgesetzt werden können und dieses auf diese Weise befähigen sollen, auch mit räumlichen Daten umgehen zu können. Ein solches Modul ist der schon erwähnte *Spatial Extender*, der von der Firma IBM in Zusammenarbeit mit ESRI als objektorientierter, räumlicher Aufsatz zur relationalen *DB2 Universal Database* entwickelt wurde. Im folgenden sollen zunächst kurz einige Grundzüge des Aufbaus und der Organisation dieses Moduls vorgestellt werden, bevor nachfolgend die Ergebnisse der durchgeführten Experimente besprochen werden.

5.3.1 Grundlagen DB2 Spatial Extender

Der *Spatial Extender* erweitert ein herkömmliches RDBMS folgendermaßen:

- Hinzufügen neuer vordefinierter, hierarchisch angeordneter (räumlicher) Datentypen mit Untertypen und Vererbung, z.B. Datentypen zur Repräsentation von Punkten, Linien und Polygonen, sowie die Möglichkeit bei Bedarf eigene Datentypen zu definieren.
- Hinzufügen neuer vordefinierter (räumlicher) Operationen, z.B. Prädikate zur Überprüfung auf Enthaltensein oder Schnitt zweier räumlicher Objekte, sowie die Möglichkeit, bei Bedarf eigene Prädikate zu definieren.
- Datenbanktypische Optimierungsmöglichkeiten für räumliche Anfragen.
- Einführen von räumlichen Indizes, um den Zugriff auf räumliche Objekte zu beschleunigen. Als Indexstruktur wird dabei standardmäßig eine Version des unter 2.4.4 vorgestellten Multilayer Grid-File verwendet. Diese ist in maximal drei Schichten organisiert, für die bei Erzeugung eines räumlichen Index die Granularität der Skalen für jede Schicht fest vorgegeben wird, wobei die unterste Schicht die feinste, die oberste Schicht die gröbste Granularität haben muß. Das Einfügen eines

räumlichen Objekts wird dann zunächst auf der untersten Schicht versucht, schneidet dieses Objekt mehr als drei dessen Skalen, so wird es auf die nächste Schicht befördert, in der es gespeichert wird, wenn nicht auch hier mehr als drei Skalen geschnitten werden. Bei Eintreten dieses Falles wird das Objekt in jedem Fall in der obersten Schicht eingefügt. Diese Methode hat den Vorteil, daß nur dann in einer Schicht gesucht werden muß, wenn nicht mehr als drei Skalen vom gesuchten Objekt geschnitten werden.

Der Zugriff auf die so erweiterte Datenbank wird über SQL (Structured Query Language), eine standardarisierte, weit verbreitete Anfragesprache für Datenbanken durchgeführt. Dies hat den Vorteil, daß jede Applikation über herkömmliche SQL-Befehle Anfragen bzw. Aktualisierungen in der Datenbank anstoßen kann, ohne dazu proprietäre Anfragesprachen benutzen zu müssen.

Der *Spatial Extender* bietet über die an eine räumlichen Datenhaltungskomponente gestellten Anforderungen zur Verwaltung von räumlichen Objekten und dem Beantworten von räumlichen Anfragen hinaus beispielsweise die Möglichkeit, räumliche Daten zu importieren und exportieren, oder mit Hilfe eines sogenannten *Geocoders* aus einer Adresse (Stadt, Straße, Hausnummer,...) eine Umwandlung dieser in eine Koordinatendarstellung durchführen zu können. Zum Importieren und Exportieren von räumlichen Daten unterstützt der *Spatial Extender* drei bekannte Industrieformate: *ESRI Shape Format, OGIS Well Known Text Format,* sowie *OGIS Well Known Binary Format.* Weiterhin unterstützt der *Spatial Extender* die Verwendung der gebräuchlichen Industriestandards *SQL3, SQL/MM* und *OGIS*.

Näheres über den *DB2 Spatial Extender* kann bei z.B. in [DAV98] nachgelesen werden.

5.3.2 Testergebnisse

Die gemäß der zweiten Entwurfsalternative entwickelte Komponente zur Verwaltung räumlicher Objekte setzt sich aus den folgenden Bausteinen zusammen:

- Methoden zum Aufbau einer Verbindung zur Datenbank.
- Methoden zum Einfügen eines Objektes.
- Methoden zum Löschen eines Objektes.
- Methoden zum Aktualisieren der Positionsinformation eines Objektes.

- Methoden zur Ausführung von Positionsanfragen.
- Methoden zur Ausführung von Bereichsanfragen.
- Methoden zur Ausführung von Nächster-Nachbar-Anfragen, sowie
- Methoden zur Beendigung der Verbindung mit der Datenbank.

Die Testumgebung, mit der die Eignung des durch den *Spatial Extender* erweiterten DB2 Datenbanksystems als Datenhaltungskomponente des Lokationsdienstes überprüft werden sollte, ähnelt der zweiten Testklasse, die für die Experimente mit den implementierten Indexstrukturen erstellt wurden. Der Zugriff auf die Datenbank wurde dabei mittels JDBC 2.0 (Java Database Connectivity) hergestellt.

Diese Klasse, `testLIDB`, ist analog zu oben zuständig für die Überprüfung der Leistungsfähigkeit der einzelnen Aktualisierungs- und Anfrageoperationen. Als Maß für die Leistungsfähigkeit der einzelnen Operationen wurde jeweils die verstrichene Zeit vom Beginn bis zum Ende eines Blockes gleicher Operationen gemessen, dies wurde mittels des DB2-Werkzeugs `db2bat` durchgeführt, das Hilfsmittel zur Stapelverarbeitung von Zugriffen auf eine DB2-Datenbank, unter anderem auch das Durchführen einer exakten Zeitmessung, bietet. Der parallele Zugriff auf die Datenbank durch mehrere Klienten wurde wieder durch Aufsetzen mehrerer Threads simuliert. Da kein eigenständiges Synchronisationsverfahren implementiert worden war, wurde diese Aufgabe dem entsprechenden (Sicherheits-)Manager der Datenbank übertragen.

Als Testrechner wurde eine Maschine mit 256 MB Hauptspeicher und zwei Pentium II Prozessoren mit je 400 Mhz Taktfrequenz ausgewählt, hauptsächlich, weil auf diesem Rechner die durch den *Spatial Extender* erweiterte DB2 Datenbank installiert war, und die Ergebnisse nicht durch die durch einen entfernten Zugriff hinzukommende Verzögerung verfälscht werden sollten.

Wie bei den implementierten Indexstrukturen wurden lediglich punktförmige, räumliche Objekte ohne Ausdehnung verwendet. Dies führte dazu, daß vom verwendeten räumlichen Index nur die unterste Ebene benutzt wurde, da ja keines der eingefügten Objekte die erforderliche Anzahl von Skalen schneiden kann, um eine Ebene nach oben befördert zu werden.

Die Tabelle, auf der alle Operationen durchgeführt wurden, wurde gemäß dem folgenden SQL-Befehl erzeugt:

```
CREATE TABLE LINFO (ObjectId VARCHAR(54), Position db2gse.ST_Point,
Old_Position db2gse.ST_Point)
```

Dadurch wurde festgelegt, daß in der ersten Spalte der eindeutige, durch eine Zeichenkette repräsentierte Bezeichner eines Objektes, in der zweiten Spalte die aktuelle Position, und in der dritten Spalte die vormals aktuelle Position des Objektes, jeweils durch den *Spatial Extender*-Datentyp für einen Punkt repräsentiert, gespeichert werden. Mit Hilfe der Werte der dritten Spalte kann beispielsweise die Richtung, in der sich ein Objekt bewegt, bestimmt werden.

Aus Leistungsgründen wurden sämtliche verfügbare Operationen „vorbereitet“ (engl.: prepared), dies bedeutet im Datenbanken-Umfeld das Parametrisieren und Vorübersetzen von häufig gebrauchten Methoden, so daß zum Ausführen nur noch konkrete Werte in die dafür vorgesehenen Parameter eingefügt werden müssen und nicht die gesamte Anweisung jedesmal komplett übersetzt werden muß.
Die Testergebnisse waren leider trotz der Inanspruchname einiger möglicher Optimierungen („Vorbereiten“ von Operationen, Benutzen von optimierten *Spatial Extender*-Anfragen,...) nicht mit den erzielten Ergebnissen für die implementierten Indexstrukturen vergleichbar. Der ermittelte Leistungsunterschied war stattdessen so deutlich, daß die Verwendung der entwickelten, auf dem *Spatial Extender* basierenden Komponente im Rahmen des Lokationsdienstes verworfen werden mußte.

Diese Entscheidung wurde noch durch einen entdeckten Fehler bekräftigt, der eine Verwendung des *Spatial Extenders* ohnehin in Frage gestellt hätte. Bei diesem handelt es sich um eine fehlerhafte Implementierung der `db2gse.ST_Distance`-Funktion, die zur Berechnung der Entfernung zweier räumlicher Objekte dient. Diese liefert aber statt einer zu erwartenden Entfernungsangabe in Metern eine Entfernungsangabe in Grad bei Verwendung eines geografischen Koordinatensystems zurück.
Rückfragen bei IBM ergaben, daß der Grund eine fehlerhafte Spezifikation innerhalb des SQL/MM Standards ist, der bei der Implementierung gefolgt wurde. Daraus resultierte, daß die Entfernungsangaben, die von der oben angegebenen Funktion zurückgegeben werden, immer nur die Einheit des unterliegenden Koordinatensystems besitzen können. Bei

geografischen Koordinatensystemen, wie z.B. dem von vielen GPS-Empfängern nativ verwendeten Systems WGS84, werden Unterschiede von Positionen zunächst einmal in der Einheit Grad gemessen und müssen noch in Meter umgerechnet werden. Das Fehlen einer solchen Umrechnung für die Entfernung zweier solcher Positionsangaben ist eine große Einschränkung für Anwendungen, vor allem wenn die Positionsinformationen etwa von GPS-Empfängern oder anderen Systemen geliefert werden, in denen geografische Koordinaten nativ verwendet werden. Weiterhin ist zu bemerken, daß die Funktion zur Berechnung des Abstands zweier Punkte beispielsweise in Nächster-Nachbar-Anfragen extrem häufig gebraucht wird, dies weitet die Auswirkungen dieser Fehlspezifikation noch aus.

Die ermittelten Leistungswerte sollen nachfolgend der Vollständigkeit halber noch angegeben werden (eine Operation pro Transaktion). Es wurden insgesamt 10 Threads aufgesetzt, die jeweils die folgenden Operationen ausführten:

- Einfügen von 10 Objekten.
- Löschen eines Objektes.
- Durchführen von 5 Bereichsanfragen mit einem Suchgebiet der Abmessungen 100 x 100 Meter.
- Durchführen von 5 Bereichsanfragen mit einem Suchgebiet der Abmessungen 1000 x 1000 Meter.
- Durchführen von 5 Nächster-Nachbar-Anfragen mit Suchradius 1000 Meter.

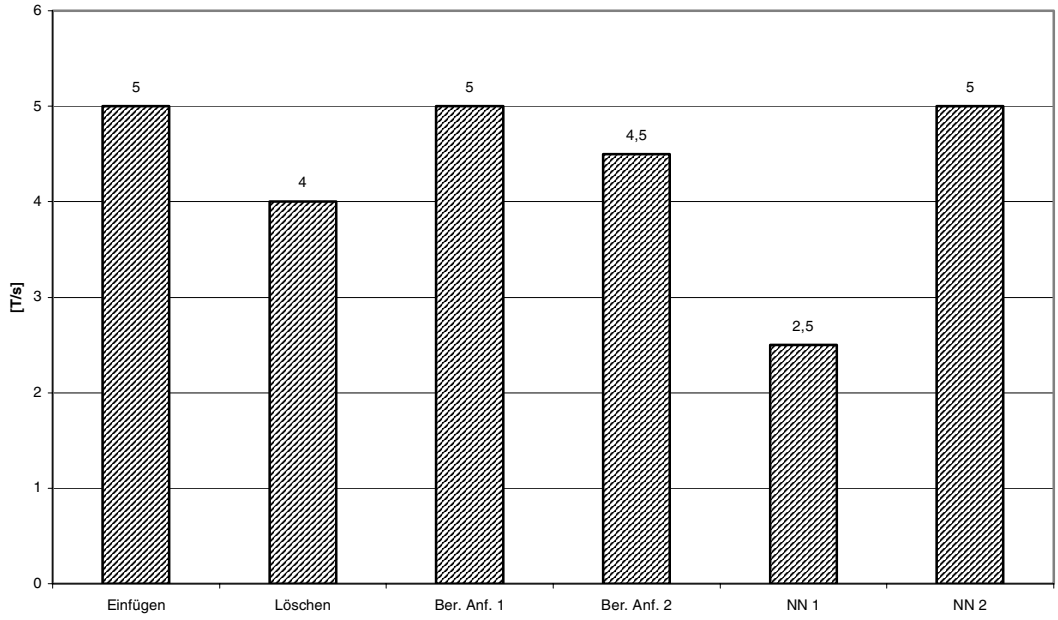


Abbildung 35: Testergebnisse mit dem IBM DB2 Spatial Extender

Interessanterweise lies sich die Leistung der Nächster-Nachbaranfragen verdoppeln, indem nicht die *Spatial Extender*-proprietäre Funktion zur Berechnung des Abstandes (`db2gse.ST_Distance`) verwendet wurde (Fall *NN1*), sondern eine selbst geschriebene, auf Anwendung des Satzes des Pythagoras basierende Funktion (Fall *NN2*). Eine Wertung dieser Tatsache sei dahingestellt.

Im nächsten Kapitel, in dem die alle erzielten Ergebnisse, sowie deren Auswirkungen auf die Wahl der Indexstruktur für die Datenhaltungskomponente des Lokationsdienstes abschließend dargestellt werden, wird noch einmal auf die möglichen Gründe für die schleche Leistungsfähigkeit des *Spatial Extenders* eingegangen.

6 Diskussion der Ergebnisse

In diesem Kapitel sollen die erzielten Ergebnisse der angestellten Untersuchungen diskutiert werden, und damit die Wahl einer der Indexstrukturen abhängig von den jeweiligen Randbedingungen, die sich im jeweiligen Anwendungsfall ergeben, begründet werden. Die folgende Tabelle vergleicht die erzielten Ergebnisse noch einmal untereinander.

	hoKd	kd20	Qd20	Gf20	Rtree
Performance Schreibende Zugriffe	4	3	1	2	5
Performance Anfragen	4	3	2	1	5
Hauptspeicherausnutzung	1	3	4	5	2
Erhaltung der Struktur	2	2	2	5	1
Einfachheit	1	2	2	4	5
Effizienz Synchronisierung	3	2	1	2	5
Verhalten bei schiefen Verteilungen	5	3	2	1	4
Gesamt	2,85	2,57	2	2,85	3,85

Tabelle 1: Vergleich der Eignung verschiedener räumlicher Indexstrukturen als Datenhaltungskomponente des Lokationsdienstes

Wichtet man all jeden Untersuchungsaspekt gleich, so zeigt sich, daß der Quad-Tree zusammengefaßt das beste Ergebnis erzielte. Die Stärken des Quad-Tree liegen demnach vor allem in der effizienten Auführung schreibender Operationen, d.h. von Einfüge- und Löschvorgängen, sowie der guten Leistungssteigerung bei Verwendung des „Write once / Read multiple“- Synchronisierungsverfahrens. Ebenfalls gut schnitt der Quad-Tree in den Bereichen „effiziente Anfragen“, „Erhaltung der Struktur“ und „Einfachheit“ ab, in denen er jeweils den zweiten Platz hinter dem homogenen kd-Baum belegte. Auch das Verhalten des Quad-Trees bei ungleichmäßigen Verteilungen war mit gut zu bewerten, wie zuvor nicht unbedingt zu erwarten war. Schwächen offenbarte der Quad-Tree bei dieser Betrachtung lediglich bei der Hauptspeicherausnutzung, hier belegte er den vorletzten Platz.

Den zweiten Platz teilten sich der heterogene kd-Baum und das Grid-File. Für den heterogenen kd-Baum bot sich ein ausgeglichenes Bild: Er landete für jede Untersuchung im Mittelfeld, hatte also keine nennenswerten Stärken und Schwächen aufzuweisen.

Das Grid-File zeigte sich für kurze Testläufe zwar gerade im Bereich „effiziente Anfragen“ allen anderen untersuchten Indexstrukturen klar überlegen, und konnte auch im Bereich „schreibende Zugriffe“ überzeugen, doch konnte es die guten Ergebnisse nicht in längeren Testläufen bestätigen. Es verdeutlichte sich hier die bekannte Schwäche des Grid-Files, nämlich das nun schon häufiger erwähnte überlineare Wachstum des Rasterverzeichnisses, das zu einer mit der Zeit sinkenden Speicherausnutzung und zu einer Vielzahl von unnötig erzeugten Zellen führt. Hieraus resultiert wiederum die sinkende Leistungsfähigkeit für die schreibenden Zugriffe und die verschiedenen Anfragetypen. Dasselbe läßt sich beim Grid-File für schiefe Datenverteilungen sagen, hier wurden für kurze Einsatzzeiträume die besten Ergebnisse erzielt, über längere Zeiträume jedoch traten wieder dieselben Probleme auf.

Hinter diesen beiden Indexstrukturen wurde die homogene Variante des kd-Baums eingeordnet. Während dieser in den Bereichen „Speicherausnutzung“ und „Einfachheit“ ganz vorne stand, mußte er sich in der Leistungsfähigkeit sowohl der schreibenden Zugriffe und der Anfragen mit dem vorletzen Platz begnügen. Abgeschlagen war der homogene kd-Baum für schiefe Objektverteilungen, in der er mit Abstand das schlechteste Ergebnis lieferte.

Der R-Baum bildet das Schlußlicht dieser Bewertung. Tatsächlich belegt dieser in nahezu allen Kategorien den letzten Platz, Vorteile konnte er sich lediglich im Bereich der Hauptspeicherausnutzung erarbeiten, in der er hinter dem homogenen kd-Baum den zweiten Platz erreichte. Problematisch ist hier aber auch der hohe absolute Speicherbedarf von ca. 90 Byte pro Objekt für die hier implementierte Version. Für schiefe Objektverteilungen kann der R-Baum nur mit seiner Leistungsfähigkeit für lesende Operationen überzeugen. Dies wird jedoch durch einen stark erhöhten Aufwand bei den schreibenden Operationen wieder relativiert, so daß sich unter dem Strich hier nur der vorletzte Platz für den R-Baum ergibt.

Es wurde daher beschlossen, als Datenhaltungskomponente für den in Kapitel 3 vorgestellten Lokationsdienst die räumliche Indexstruktur Quad-Tree zu verwenden, da diese über alle Kategorien hinweg zusammengefaßt die besten Ergebnisse erzielte. Diese hat dann den in untenstehender Abbildung skizzierten Aufbau:

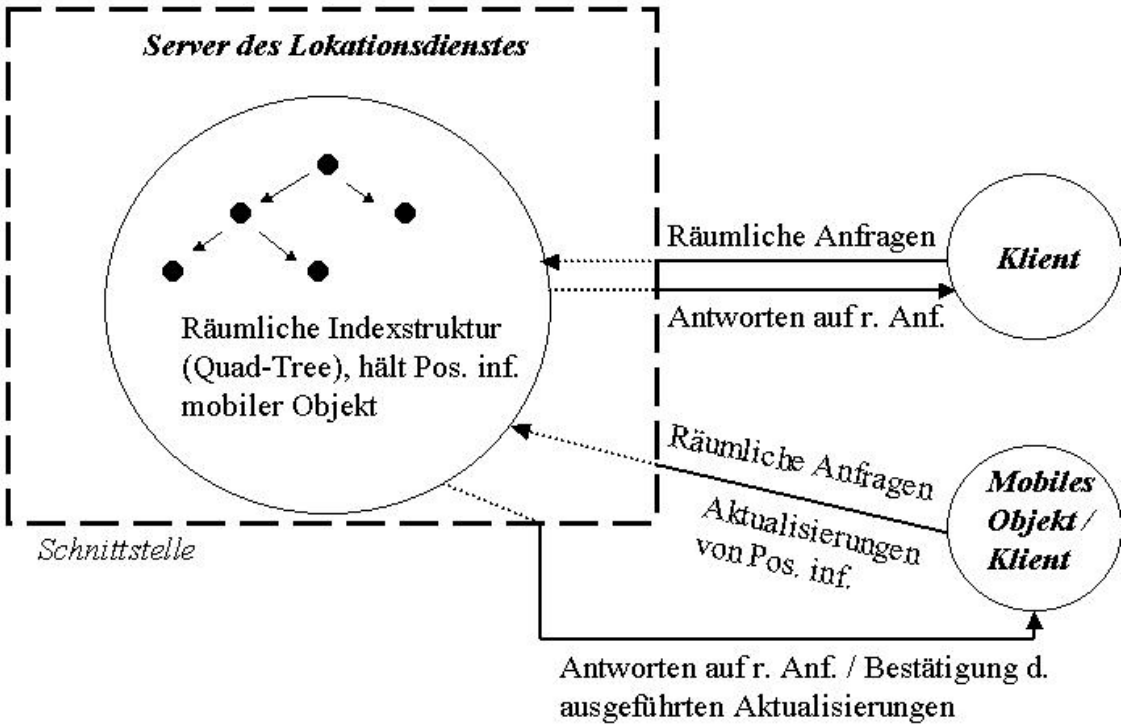


Abbildung 36: Aufbau und Verwendung der Datenhaltungskomponente(LIStorageMainMemory) des Lokationsdienstes

Wie aus der Abbildung ersichtlich ist, werden über definierte Schnittstellen Operationen auf einer den Kern der Komponente bildenden Indexstruktur ausgeführt. Initiatoren dieser Operationen können sowohl mobile Objekte im Sinne des Lokationsdienstes, als auch stationäre Klienten sein. Die Komponente bietet zur Zeit folgende Funktionalität:

- Einfügen, Löschen und Aktualisieren von räumlichen Objekten, bzw. deren Positionsinformationen.
- Beantworten von Positions-, Bereichs- und Nächster-Nachbar-Anfragen.
- Synchronisation paralleler, konkurrierender Zugriffe.

Natürlich kann auf der Basis der hier angestellten Untersuchungen je nach Situtation auch eine andere Indexstruktur als Datenhaltungskomponente gewählt werden. Denkbar sind in diesem Zusammenhang z.B. folgende Konstellationen:

- Wenn begrenzter Hauptspeicher für einen Server des Lokationsdienstes zur Verfügung stehen sollte, wäre die Verwendung des (homogenen) kd-Baums denkbar.

- Für kurze Einsatzzeiträume mit hohen Anforderungen an die Leistungsfähigkeit für Anfragen und Aktualisierungen ist das Grid-File empfehlenswert. Denkbar wäre hier für längere Einsatzzeiträume auch das Löschen des gesamten Rasterverzeichnisses, gefolgt von der Neuerzeugung eines solchen und dem Einfügen aller Objekte, die im alten Rasterverzeichnis gehalten wurden. Dieser Vorgang könnte ausgelöst werden, wenn der Betrag der Hauptspeicherausnutzung eine gewisse wählbare Schranke unterschreitet, ein sicheres Zeichen für eine fortgeschrittene Entartung des Grid-Files, bzw. dessen zugeordnetem Rasterverzeichnisses.
- Jede der hier vorgestellten Indexstrukturen, bis auf den R-Baum, wurde vornehmlich für das Speichern von nur punktförmigen räumlichen Objekten entworfen. Das Verwalten von Objekten mit räumlicher Ausdehnung wird dagegen nicht direkt von diesen unterstützt, sondern muß durch eine der unter 2.4.4 genannten Methoden, etwa durch Transformation, implementiert werden. Hier könnte der R-Baum sein Anwendungsgebiet haben, da dieser in der Klasse der SAMs eingeordnet wird und damit ausdrücklich für die Verwaltung von räumlich ausgedehnten Objekten entworfen wurde. In diesem Fall müssten neue Experimente mit den erweiterten Indexstrukturen und dem R-Baum durchgeführt werden, die unter Umständen zu ganz anderen Ergebnissen führen könnten.

Dieses Kapitel abschließen soll noch kurz eine Einschätzung der Ergebnisse der Experimente mit dem *IBM DB2 Spatial Extender*. Wie bereits erwähnt, konnten diese nicht im entferntesten an die erzielten Resultate der untersuchten Indexstrukturen heranreichen. Mögliche Gründe für dieses Verhalten können sein:

- Die verwendete Hardware: Wie bereits gesagt, wurde ein Rechner mit 256 MB Hauptspeicher und 2 Pentium II Prozessoren mit je 400 MHz als Testmaschine verwendet. Hier könnte vor allem die Größe des Hauptspeichers der die Testergebnisse entscheidend verschlechternde Flaschenhals sein. Interessant wären in diesem Zusammenhang etwa Tests mit dem *Spatial Extender* auf größeren Maschinen, die dementsprechend mit mehr Hauptspeicher ausgestattet sind.
- Der Zugriff über JDBC. Der Zugriff auf eine Datenbank über JDBC ist für den Benutzer sehr komfortabel und macht viele unterliegende Vorgänge transparent. Dies könnte zu Lasten des Durchsatzes gehen. Denkbar ist auch eine gewisse Ineffizienz

des verwendeten JDBC-Treibers, der Umwandlungen von JAVA-Datentypen in das entsprechende Datenbankformat und zurück durchführen muß.

- Das grundlegende Problem, daß sich bei der Umwandlung von Objekten in eine für relationale Datenbanken verwendbare Form und zurück ergibt. Da der *Spatial Extender* nur ein (objektorientierter, räumlicher) Aufsatz auf ein an sich relationales Datenbanksystem ist, muß diese Umwandlung aber bei jeder Einfügung, Aktualisierung und Anfrage durchgeführt werden. Zu vermuten ist, daß bei diesen Umwandlungen aufwendige Operationen auf den Tabellen der relationalen Datenbank nötig sind, die das Erreichen einer hohe Leistungsfähigkeit verhindern könnten.

Im letzten Kapitel soll nun abschließend ein kurzer Überblick über die im Rahmen dieser Arbeit durchgeführten Tätigkeiten, sowie ein Ausblick auf mögliche nachfolgende Forschungs- und Entwicklungsarbeiten in diesem Themengebiet gegeben werden.

7 Zusammenfassung und Ausblick

Diese Arbeit hatte als Zweck die Untersuchung von sogenannten räumlichen Indexstrukturen im Hinblick auf die Eignung als Datenhaltungskomponente für den Lokationsdienst der NEXUS-Plattform an der Universität Stuttgart. Dieser Lokationsdienst ermöglicht es den bei ihm registrierten Klienten, Antworten auf Anfragen der Art „Finde das zu mir nächstgelegene registrierte Objekt“, oder „Finde alle registrierten Objekte in einem Gebiet“ zu erhalten. Um diese Anfragen, und auch die eventuell häufig nötigen Positionsaktualisierungen effizient ausführen zu können, muß die Komponente, die für das eigentliche Halten der Positionsinformationen zuständig ist, speziell für die Verwaltung von mehrdimensionalen Daten entworfen worden sein. In der Vergangenheit wurde schon intensive Forschungsarbeit für den Entwurf solcher Indexstrukturen geleistet, dies führt zu einer immens großen Zahl an unterschiedlichen Vorschlägen, die sich oft nur in Details voneinander unterscheiden. Weiterhin gibt es keine allgemein „beste“ Indexstruktur, die für jeden Anwendungszweck gleich gut geeignet ist. Vielmehr eignen sich manche Indexstrukturen besonders für bestimmte Anwendungsbereiche, liefern dafür in anderen Gebieten wieder schlechtere Ergebnisse.

Zunächst mußten also die vom erwähnten Lokationsdienst geforderten Eigenschaften einer Datenhaltungskomponente erarbeitet werden und einige Indexstrukturen dafür ausgewählt werden. Da das Durchführen von Experimenten mit diesen Strukturen im Hinblick auf die erarbeiteten Eigenschaften des Lokationsdienstes ein großer Bestandteil dieser Arbeit sein sollte, mußten die ausgewählten Indexstrukturen nachfolgend implementiert werden. Als Implementierungssprache wurde JAVA (Version 1.3) gewählt.

Die Konzeption des Lokationsdienstes macht das Vorhandensein einer Sekundärspeicherverwaltung bei den Indexstrukturen unnötig, dementsprechend sollte die Implementierung einer Indexstruktur die Daten nur im Hauptspeicher ablegen. Eine zweite, zu untersuchende Variante war das Verwenden einer kommerziellen, auf einer herkömmlichen Datenbank basierenden Lösung anstelle der hauptspeicherbasierten Alternative. Hier wurde der *IBM Spatial Extender* untersucht, ein objektorientierter Aufsatz auf das RDBMS *IBM DB2 UDB* zum Verwalten räumlicher Daten.

Nachdem die Indexstrukturen ausgewählt und implementiert wurden, wurde ihre Eignung auf Tauglichkeit für den Lokationsdienst empirisch und in Experimenten untersucht, gleichzeitig wurden auch Experimente mit der kommerziellen Lösung durchgeführt.

Die Ergebnisse der Untersuchungen sind in Kapitel 6 zusammengefaßt, hier wurde auch eine Empfehlung abgegeben, welche der untersuchten Alternativen sich speziell für den Lokationsdienst am besten zu eignen scheint.

Durch die schon weiter oben angesprochene starke Spezialisierung von räumlichen Indexstrukturen auf gewisse Anwendungsgebiete lassen sich über den Rahmen dieser Arbeit hinaus zahlreiche Ideen für weitergehende Forschungen formulieren.
So wurde der Begriff „räumlich“ in dieser Arbeit vereinfachend als zweidimensionaler Fall angesehen, d.h. ein räumliches Objekt ist in seiner Position durch die Angabe zweier Werte festgelegt. Weiterentwicklungen des Lokationsdienstes können es aber nötig machen, daß Objekten auch in einer dritten Dimension unterscheidbar sind, z.B. einer Höhenangabe. Das Hinzukommen einer dritten Dimension erfordert die Erweiterung der bisher implementierten Indexstrukturen. Eine solche Erweiterung hat voraussichtlich Auswirkungen auf die erzielten Testergebnisse und könnte zu einem von diesen abweichenden Ergebnis führen. Hier sei nur angedeutet, daß vor allem der kd-Baum sehr gut geeignet scheint, im dreidimensionalen Fall eingesetzt zu werden. Dies begründet sich auf der Eigenschaft des kd-Baums, bei einer Baumtraversierung, die praktisch allen Operationen zugrunde liegt, den weiter zu verfolgenden Pfad durch den Vergleich nur einer Komponente des dort gehaltenen Referenzobjektes zu bestimmen. Der dreidimensionale Fall läßt sich also einfach so implementieren, daß nun drei statt zwei Vergleichsalternativen abhängig von der Höhe des Knotens im Baum berücksichtigt werden müssen. Gleichzeitig wird sich die Hauptspeicherausnutzung kaum verschlechtern, denn nach wie vor existieren nur jeweils zwei Nachfolgerzeiger in jedem Knoten. Beim Quad-Tree fallen die Vergleiche im dreidimensionalen Fall dafür beispielsweise umso teurer aus, denn nun muß eine aus 8 Alternativen ausgewählt werden, um den richtigen Nachfolger zu bestimmen. Gleichfalls müssen nun 8 Nachfolgerzeiger gehalten werden, was zu einer noch schlechteren Hauptspeicherausnutzung führen dürfte.

Interessant wäre auch die Implementierung von dynamischen Reorganisationsmaßnahmen für die Indexstruktur Grid-File, deren Leistungsfähigkeit ja stark unter dem überlinearen

Wachstums des Rasterverzeichnisses leidet. Geeignet scheint hier das sogenannte *Buddy*-Verfahren, daß schlecht besetzte, benachbarte Zellen miteinander verschmilzt und so zu einer besseren Speicherausnutzung und einer daraus resultierenden Leistungskonservierung auch über längere Einsatzzeiträume kommen könnte.

Über die den Kern der Datenhaltungskomponente betreffenden zukünftigen Forschungen hinaus wäre auch eine Erweiterung der gesamten Komponente denkbar. Interessant könnte in diesem Zusammenhang beispielsweise das Implementieren von Methoden für das Auslösen von „Aufräumprozessen“ sein. Dabei ist z.B. das ständige, bzw. periodische Überwachen der Indexstruktur im Bezug auf deren Aufbau gemeint. Eine Überlegung wert wäre hier das das Erzeugen einer neuen Instanz dieser Indexstruktur, wenn die bisherige Instanz eine derart ungünstige Gestalt angenommen hat (z.B. stark ungleichmäßige Struktur), daß die Leistungsverluste nicht mehr tolerierbar sind. Die dann im voraus bekannten Objekte der bisherigen Instanz können nun so eingefügt werden, daß eine ausbalancierte Struktur entsteht. Dadurch könnte dann das Zeitverhalten nachfolgender Zugriffe stark verbessert werden.

Anhang A: Schnittstellen der Datenhaltungskomponente

Methoden zur Verwaltung der Datenhaltungskomponente:

Erzeugen einer räumlichen Indexstruktur mit Spezifikation des abgedeckten Gebietes und einer maximalen Blattkapazität.
public void init(Segment area, int nodeSize);

Beenden der Datenhaltung mit einer räumlichen Indexstruktur.
public void close();

Methoden für schreibende Zugriffe auf der räumlichen Indexstruktur:

Einfügen eines neuen Objektes mit Spezifikation dessen eindeutiger Kennung, der aktuellen und der bisherigen Position.
public void newObject(ObjectId oid, Sighting pos, Sighting lastpos);

Löschen eines Objektes mit der spezifizierten eindeutigen Kennung aus der räumlichen Indexstruktur.
public void removeObject(ObjectId oid);

Aktualisieren der Positionsinformation des Objektes mit der spezifizierten eindeutigen Kennung auf die spezifizierte Position.
public void setPosition(ObjectId oid, Sighting pos);

Methoden für Anfragen an die Datenhaltungskomponente:

Anfragen, ob Objekt mit der spezifizierten eindeutigen Kennung in Datenhaltungskomponente gespeichert ist.
public boolean existsObject(ObjectId oid);

Herausfinden der aktuellen Position des Objektes mit der spezifizierten eindeutigen Kennung.
public Sighting getPosition(ObjectId oid);

Herausfinden der bisherigen Position des Objektes mit dieser eindeutigen Kennung.
public Sighting getLastPosition(ObjectId oid);

Ausführen einer Positionsanfrage mit Spezifikation der Position des zu suchenden Objektes.
public ObjectId positionQuery(Coordinate pos);

Ausführen einer Nächster-Nachbar-Anfrage für das Objekt an der spezifizierten Position mit Spezifikation des maximalen Suchradius.
public Sighting nearestObject(Coordinate pos, double maxDist);

Ausführen einer Bereichsanfrage mit Spezifikation des Suchbereiches.
public Vector objectsInArea(Area a);

Anhang B: Literaturverzeichnis

[BAN95] Banks, D., M. Kornacker and M. Stonebraker (1995): *High Concurrency Locking in R-Trees.* In the VLDB Journal, pp. 134-145

[BEC90] Beckmann, N., H. P. Kriegel, R.Schneider and B. Seeger (1990): *The R*-Tree: An efficient and robust access method for points and rectangles.* In Proc. ACM SIGMOD Int. Conf. On Management of Data, pp. 322-331

[BEN74] Bentley, J. L. and R. A. Finkel (1974): *Quad-Trees: A Data Structure on Retrieval on Composite Keys.* Acta Informatica 4, pp. 1-9

[BEN75] Bentley, J. L. (1975): *Multidimensional binary search trees used for associative searching*. In Comm. Of the ACM 18(9), pp. 509-517

[BEN79] Bentley, J. L. (1979): *Multidimensional binary search in database applications.* IEEE Trans. Software Eng. 4(5), pp. 333-340

[BER96] Berchthold, S., D. Keim and H. P. Kriegel (1996): *The X-Tree: An Index Structure for High-dimensional data.* In Proc. 22nd Int. Conf. On Very Large Databases, pp. 28-39

[BMC72] Bayer, R. and E. M. McCreight (1972): *Organization and Maintenance of large ordered indices.* Acta Informatica 1(3), 173-189

[DAV98] Davis, J. (1998): *The IBM DB2 Spatial Extender: Managing Geo-Spatial information within the DBMS.* IBM Corp., http://www-4.ibm.com/software/data/pubs/papers/spatial/

[ERI98] Erickson, J. (1998): *Geometric Data Structures: Lectures and References.* http://compgeom.cs.uiuc.edu/~jeffe/teaching/497f98/lectures.html

[FAG79] Fagin, R., J. Nievergelt, N. Pippenger and R. Strong (1979): *Extendible Hashing: A fast access method for dynamic files.* ACM Trans. Database Systems 4(3), 315-344

[FRE97] Merkle, F. (1997): *Fred's kd-Tree Research Project.* http://www.me.unm.edu/~bgreen/QEM97/FRED/KD-TREES.HTM

[FRI77] Friedman, S. H., Bentley J. L. and Finkel, R.A. (1977): *An Algorithm for finding best matches in Logarithmic Expected Time.* ACM Trans. On Mathematical Software, 3, 209-226

[GAE98] Gaede, V. and O. Günther (1998): *Multidimensional Access Methods.* In ACM Computing Surveys Vol. 30, n. 2, pp. 170-231

[GUT84] Guttman, A. (1984): *R-Trees: A Dynamic Index Structure for Spatial Searching.* In Proc. ACM SIGMOD ´84, pp. 47-57

[HÄR99] Härder, T. and E. Rahm (1999): *Datenbanksysteme: Konzepte und Techniken der Implementierung.* Springer, ISBN 3-540-65040-7

[HIN98] Hinterberger, H. and B. Bauer-Messner, ETH Zürich, (1998): *Discrete Object Detection and Motion Registration Based on a Data Management Approach.* In IEEE Proc. Of SSDBM '98 (Capri/Italy)

[KAF94] Kamel, I. and C. Faloutsos (1994): *Hilbert R-Tree: An improved R-Tree using fractals.* In Proc. 20th Int. Conf. On Very Large Databases, pp. 500 – 509

[KAN97] Kanth K. V., D. Serena and A. K. Singh (1997): *Improved Concurrency Control Techniques for Multidimensional Index Structures.* 11th International Parallel Processing Symposium

[KRI84] Kriegel, H. P. (1984): *Performance comparison of index structures for multikey retrieval*. In Proc. ACM SIGMOD Int. Conf. On Management of Data, pp. 186-196

[LEH81] Lehmann P. And S. Yao (1981): *Efficient Locking for Concurrent Operations on B-Trees.* ACM TODS

[LOC01] Leonhardi, A. and K. Rothermel (2001): *Architecture of a Large-scale Location Service,* Universität Stuttgart, Fakultät Informatik, Fakultätsbericht Nr. 2001/01.

[MAR96] Margetts, S. (1996): *Nearest Neighbor Queries using kd-Trees.* Technical Report Cardiff University, MathSource Doc. no.: 0208-471

[MOO91] Moore, A. W. (1991): *An Introductory Tutorial on kd-Trees.* Extract from Efficient Memory-based learning for Robot Control (Tech. Rep. 209), Comp. Lab., University of Cambridge

[NEX99] Hohl, F., U. Kubach, A. Leonhardi, K. Rothermel and M. Schwehm (1999): *Next Century Challenges: Nexus - An Open Global Infrastructure for Spatial-Aware Applications*. Proceedings of the Fifth Annual ACM/IEEE International Conference on Mobile Computing and Networking (MobiCom'99), Seattle, Washington, USA

[NGU98] Nguyen, D., K. DuPrie and P. Zografou (1998): *A Multidimensional Binary Search Tree for Star Catalog Correlations.* Smithsonian Astrophysical Observatory, Cambridge, MA 02138

[NIE84] Nievergelt, J., H. Hinterberger and K. Sevcik (1984): *The Grid-File: An adaptable, symetric, multikey file structure*. ACM Trans. Database Systems 9(1), pp. 38-71

[OGI01] Website Open GIS Consortium (2001): http://www.opengis.org

[REG85] Regnier, M. (1985): *Analysis of Grid File Algorithms.* BIT 25, pp. 335-357

[ROT99] Rothermel, K. (1999): *Skript zur Vorlesung "Verteilte Systeme"*, Universität Stuttgart, IPVR, Abteilung Verteilte Systeme

[SAM84] Samet, H. (1984): *The Quadtree and Related Hierarchical Data Structures.* ACM Computing Surveys 16(2), pp. 187-260

[SIX88] Six, H. and P. Widmayer (1988): *Spatial Searching in Geometric Databases.* In Proc. 4th IEEE Int. Conf. On Data Eng., pp. 496-503