

# Verwaltung geographischer Daten mit Hilfe eines Add-ons für Standard-Datenbanken

Jörg Roth

Fakultät für Informatik  
Ohm-Hochschule Nürnberg  
Kesslerplatz 12, 90489 Nürnberg  
Joerg.Roth@Ohm-Hochschule.de

**Abstract:** Für die Verwaltung großer Mengen geographischer Daten setzen sich zunehmend Datenbankerweiterungen für relationale Datenbanken durch. Aufgrund der besonderen Rolle geometrischer und geographischer Datenstrukturen muss der Zugriff einer Anwendung auf Datenbankfunktionen signifikant erweitert werden. In diesem Papier wird ein Ansatz vorgestellt, bei dem die räumliche Erweiterung nicht als Bestandteil der Datenbank realisiert wird, sondern als zusätzliche Software-Bibliothek verstanden wird, die auf einer beliebigen Standard-Datenbank operieren kann. Neben einem geeigneten räumlichen Index zur Beschleunigung der Abfragen, wird eine objektorientierte Schnittstelle zur Nutzung der räumlichen Funktionen angeboten. Das Papier schließt mit Performance-Betrachtungen des Ansatzes ab.

## 1 Einleitung

Der Ort ist neben der Zeit einer der wichtigsten Daten, die den Kontext eines Benutzers beschreiben. Kennt man die geographische Position eines mobilen Anwenders, kann man über Anfragen an Geo-Datenbanken verschiedene Informationen über die Situation des Benutzers ermitteln. Die Speicherung geometrischer und ortsbezogener Daten in relationalen Datenbanken stellt jedoch ein Problem dar, da Geometrien, je nach Sichtweise, sowohl die Eigenschaft eines Spaltenattributs, als auch die Eigenschaft einer Relation besitzen. Als Lösung werden räumliche Erweiterungen zu Datenbanken angeboten, die Geometrien als zusätzliche Datentypen betrachten, auf die spezielle geometrische Operationen und Abfragen angewendet werden können (Abb. 1a).

Für räumliche Datenbanken wurden zwar Standards vorgeschlagen, jedoch haben die verschiedenen Datenbankhersteller diese in unterschiedlichem Maße umgesetzt, darüber hinaus eigene, proprietäre Eigenschaften hinzugefügt. In diesem Papier wird daher ein anderer Ansatz vorgeschlagen: ein so genanntes *Geospatial Add-on* wird als Bibliothek zur Anwendung hinzugebunden und erlaubt ihr, Geometrien zu speichern und geometrischen Anfragen zu formulieren (Abb. 1b). Die eigentliche Datenbankfunktionalität wird aber auf einer relationalen Datenbank *ohne* räumliche Erweiterung ausgeführt.

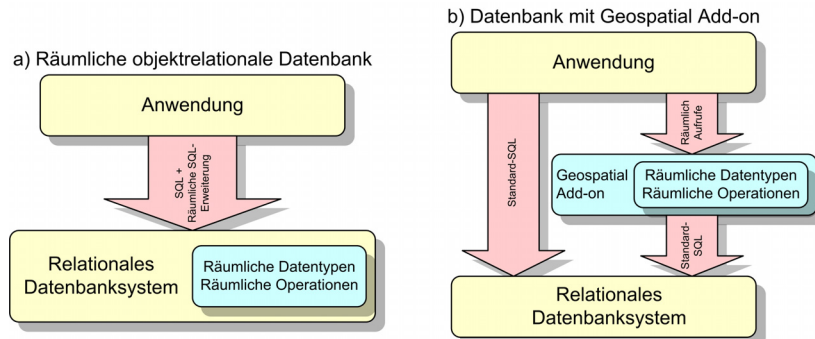


Abbildung 1: Varianten zur Verwaltung räumlicher Daten

Die Vorteile dieser Vorgehensweise:

- Zurzeit verwenden viele ortsbezogene Anwendungen und Dienste noch Standard-Datenbanken, indem ausschließlich Punkt-Geometrien (z.B. Points of Interests oder Koordinaten von Benutzern) gespeichert werden. In diesem Sonderfall können geometrische Anfragen (z.B. über die Entfernung von Punkten) über Standard-SQL abgewickelt werden. Der hier beschriebene Ansatz erlaubt eine nahtlose Migration solcher ortsbezogenen Anwendungen in Richtung komplexer Geometrien ohne große Modifikationen.
- Das unterlegte Datenbankmanagementsystem kann ohne Änderung der Anwendung ausgetauscht werden. So kann eine Anwendung beispielsweise prototypisch erst mit einem Datenbankmanagementsystem mittlerer Leistungsfähigkeit entwickelt werden, das später durch ein leistungsfähigeres Datenbankmanagementsystem ersetzt werden kann, ohne die Anwendung modifizieren zu müssen.
- Da das Add-on im Speicher der Anwendung ausgeführt wird, stehen die räumlichen Daten direkt der Anwendung als Objekte zur Verfügung. Damit übernimmt das Add-on das so genannte *Objektrelationale Mapping (ORM)*, ohne dass man hierzu ein weiteres Rahmenwerk integrieren muss.
- Mittlerweile existieren Datenbanksysteme auch auf mobilen Endgeräten. So gibt es z.B. für die Endgeräteplattformen Symbian OS, Android und iPhone Realisierungen von SQLite. Geodatenbanken sind dagegen typischerweise noch nicht für mobile Endgeräte portiert. Mit dem vorgeschlagenen Add-on können Anwendungen direkt auf dem Endgeräte auch räumliche Datentypen verwenden.

In diesem Papier wird das Add-on beschrieben, das innerhalb des HomeRun-Projektes entwickelt wurde. Es verwendet einen eigenen räumlichen Index, den *Extended Split Index* und stellt eine Objekt-Schnittstelle für die Formulierung räumlicher Anfragen zur Verfügung.

## 2 Verwandte Arbeiten

Im Datenbank-Bereich gibt es verschiedene räumliche Erweiterungen, die direkt in ein Datenbankmanagementsystem integriert wurden, z.B. *Oracle Spatial* [Mur08], *PostgreSQL/PostGIS* [Neu09] oder *MySQL Spatial Extensions* [Sun09]. Gegenüber den Standard-Datenbanken stellen die räumlichen Varianten zusätzlich geometrische Datentypen, geometrische Operationen (inkl. der Formulierung geometrischer Bedingungen) und einen räumlichen Index zur Beschleunigung räumlicher Abfragen zur Verfügung. Als räumliche Indizes kommen häufig Verfahren zum Einsatz, die Geometrien baumartig organisieren, beispielsweise Quadrees [FB74] oder verschiedene Varianten der R-Trees [Gut84]. Die Verwaltung des Indexes wird vom Anwendungsentwickler weitgehend verborgen.

Die Schnittstelle der Anwendung zur Datenbank ist typischerweise SQL, durchgereicht zur Anwendung in Form von APIs wie ODBC und JDBC. Einen weiteren Komfort liefern Persistenz-Rahmenwerke wie Hibernate, bzw. Hibernate Spatial [Hib09], die die Inhalte der Datenbank direkt in den Objektraum der Anwendung einblenden. Hibernate Spatial verwendet so genannte *Provider*, um den Zugriff zur räumlichen Datenbank zu kapseln. Räumliche Abfragen werden mit einer SQL-Erweiterung HQL formuliert. Allerdings werden die Eigenschaften der räumlichen Datenbank nur durchgereicht, so dass keine vollständige Unabhängigkeit von dem unterlegten Datenbanksystem erreicht wird.

Zur Standardisierung räumlicher Erweiterungen und dem Zugriff über SQL hat das *Open Geospatial Consortium (OGC)* Vorschläge gemacht. Die so genannten *Simple Features* geben ein Rahmenwerk für Geometrien vor [Herr05], indem Geometrie-Klassen wie z.B. *LineString* oder *MultiPolygon* und deren Abhängigkeiten definiert werden. In einem weiteren Dokument [Herr06] wird darüber hinaus festgelegt, wie diese Klassen auf SQL abgebildet werden. Einen ähnlichen Ansatz verfolgt ISO SQL/MM Spatial [Sto03], der auch auf den OGC Simple Features basiert. Ein augenfälliger Unterschied zu dem OGC-Vorschlag ist, dass alle räumlichen Funktionen mit `ST_` beginnen, darüber hinaus definiert SQL/MM Spatial einige weitere räumliche Funktionen.

Die existierenden räumlichen Datenbanken implementieren in unterschiedlichem Maße die durch die Standards vorgegebenen Eigenschaften [Bri07] – so fehlen in einigen Systemen bestimmte Eigenschaften oder entsprechen nicht den Vorgaben. Darüber hinaus sind der Satz geometrischer Funktionen sowie die Schnittstelle zum Zugriff auf die Geometrien noch sehr unterschiedlich ausgeführt. Als Beispiel: PostGIS verwendet die Funktion `AddGeometryColumn`, um eine Geometrie-Spalte anzulegen; Werte werden mit `GeometryFromText()` zugewiesen. In MySQL Spatial Extensions gibt es dagegen den Spaltentyp `GEOMETRY`, der mit `CREATE TABLE` angelegt werden kann. Werte werden mit `GeomFromText()` definiert. Oracle Spatial schließlich führt den Spaltentyp `SDO_GEOMETRY` ein, der mit `SDO_GEOMETRY(...SDO_ORDINATE_ARRAY())` mit Werten versorgt werden kann. Weitere Unterschiede in der Schnittstelle zur Anwendung ziehen sich durch die gesamten räumlichen Funktionalitäten. Bei der Anwendungsentwicklung muss man sich daher früh auf ein bestimmtes Datenbankmanagementsystem festlegen.

### 3 Das HomeRun-GAO

Ziel des HomeRun-Projektes ist, Standard-Komponenten für ortsbezogene Dienste zu entwickeln. Hierbei soll insbesondere die Realisierung von Diensten außerhalb des Massenmarktes für überschaubare Benutzerzahlen signifikant vereinfacht werden. Solche Dienste sind aktuell in der Entwicklung noch zu teuer, da die Kosten nicht durch hohe Benutzerzahlen kompensiert werden können. HomeRun wird eine Reihe notwendiger, häufig wiederkehrender Basisfunktionen zur Verfügung stellen, beispielsweise die effiziente Verwaltung räumlicher Daten auf Dienst-Servern oder die Kartendarstellung und Positionsbestimmung auf Endgeräten.

Eine der wesentlichen Komponenten für die Datenhaltung ist die Komponente *HomeRun-GAO* (*Geospatial Add-on*) als Alternative für eine räumliche Datenbank (Abb. 2).

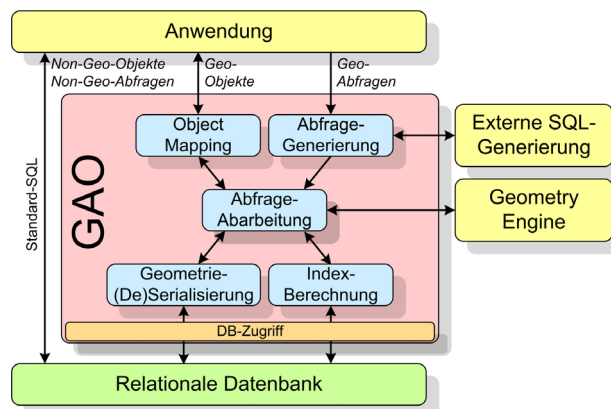


Abbildung 2: Struktur des HomeRun-GAO

Die Komponente hat folgende Eigenschaften:

- Die Funktionalitäten werden über eine objektorientierte Klassenbibliothek bereitgestellt (zurzeit in Java), die der Dienst oder die Anwendung nutzen kann. Diese ist mit ca. 80kB Binärcode extrem schlank.
- Ein neuartiger räumlicher Index genannt *Extended Split Index* erlaubt die geometrische Vorfilterung von Abfragen.
- Ein Serialisierungsmechanismus erlaubt die Ablage von Geometrien in einer Datenbankspalte. Zusätzlich wird die Index-Information in einer einzelnen weiteren Datenbankspalte gespeichert. Die GAO-Komponente stellt ein automatisches Objekt-Mapping für die Anwendung zur Verfügung.
- Die Bearbeitung der Geometrien geschieht ausschließlich im Objektraum der Anwendung. Hierzu wird eine externe *Geometry-Engine* auf der Basis der Java Topology Suite JTS [Aq03] verwendet, die vollständig den Satz der OGC Simple Fea-

tures implementiert. Geometrische Abfragen werden komfortabel über Funktionen zusammengestellt und automatisch auf Standard-SQL-Abfragen abgebildet.

- Die trotz Standardisierung vorhandenen Unterschiede verschiedener Datenbanksysteme werden in einem einzelnen Schnittstellen-Objekt (*DB-Zugriff*) behandelt. Zurzeit existieren DB-Zugriffe für die Datenbanken PostgreSQL, MySQL, Oracle, Microsoft SQL-Server, SQLite und HSQLDB. Weitere Schnittstellen können ohne großen Aufwand realisiert werden.

Typische räumliche Abfragen an das GAO ermitteln Datenreihen, die bestimmte räumliche Bedingungen erfüllen. Beispiele: *Welche Geo-Objekte umschließen einen bestimmten Punkt? Welche Geo-Objekte liegen vollständig in einem gegebenen Polygon? Welche schneiden eine bestimmte Linie?* Als wesentliche Einschränkung können zurzeit keine Joins zweier Datenbanktabellen auf der Basis ihrer Geometrien durchgeführt werden, d.h. der eine Teil einer binären geometrischen Bedingung muss durch die anfragende Anwendung definiert werden. In der Praxis sind die verfügbaren Möglichkeiten für typische ortsbezogene Dienste völlig ausreichend. Hier werden beispielsweise alle Geo-Objekte für die Kartendarstellung ermittelt, die in einem bestimmten Kartenausschnitt liegen, oder es werden Objekte abgefragt, die auf einer bestimmten Benutzerposition liegen. Solch gestaltete Abfragen können durch das GAO bequem und effizient ausgeführt werden.

### 3.1 Der Extended Split Index

Auf einer baumartigen Struktur basierende räumliche Indizes haben einen signifikanten Nachteil: durch Einfügung, Änderung oder Löschung sind nicht nur Index-Werte der modifizierten Datenzeile betroffen – potentiell können durch eine Baum-Reorganisation die Index-Werte *aller* Zeilen einer Tabelle geändert werden. Für ein Add-on ist diese Herangehensweise kritisch, da es die Index-Werte über die Standard-Schnittstelle zur Datenbank modifizieren müsste. Während eine in die Datenbank integrierte räumliche Komponente einen effizienteren (da internen) Zugriff auf die Index-Strukturen hat, muss ein externes Add-on die Index-Werte wie eine typische Datenbankspalte betrachten. Großflächige Änderungen würden damit große Ausführungszeiten verursachen.

Ein weiterer Nachteil baumartiger Indizes ist, dass eine Suche durch das Abwandern eines Baumes realisiert wird, was mehrere Tabellenzugriffe erfordern würde. Für das Add-on wurde daher ein neuartiger räumlicher Index entwickelt, bei dem ein Index-Wert ausschließlich von der jeweiligen Datenzeile abhängt. Es ist in keinem Fall eine Reorganisation mehrerer Datenzeilen notwendig.

Der Index basiert auf der Idee des *Split Index* [Bey09] und wurde in der aktuellen Fassung zum *Extended Split Index* weiterentwickelt. Als Grundidee wird die zweidimensionale Lage einer Geometrie auf eine kleine Menge eindimensionaler Index-Wert abgebildet, die effizient über Intervall-Suchen der Datenbank recherchierbar sind. Die zweidimensionale Suche wird dadurch auf etablierte Mechanismen, insb. auf eindimensionale Index-Spalten innerhalb der Standard-Datenbanken abgebildet.

Wichtig ist bei dieser Index-Struktur, wie generell bei Indizes, dass die Menge der Kandidaten, die eine bestimmte geometrische Eigenschaft erfüllen, schon bei der ersten Abfrage signifikant eingeschränkt werden kann. Die Liste der Kandidaten muss außerhalb der Datenbank durch die Geometry-Engine exakt geometrisch überprüft werden. Die Feinfilterung der Kandidatenmenge geschieht im Objektraum der Anwendung.

Der Extended Split Index erlaubt die Verwaltung eines *endlichen* zweidimensionalen Bereiches  $(x_0 \dots x_{max}, y_0 \dots y_{max})$ . Die Erweiterung auf mehr Dimensionen ist denkbar, wird hier aber nicht vorgestellt. Zur Speicherung weltweiter geographischer Daten würde man Längengrad-Breitengrad-Koordinaten verwenden.

Die Grundidee ist, ganzen Zahlen  $z$  eindeutig bestimmte Teilflächen (genannt *Kacheln*) des Bereiches zuzuordnen. Die Zuordnung wird in Abb. 3 (unten) illustriert.

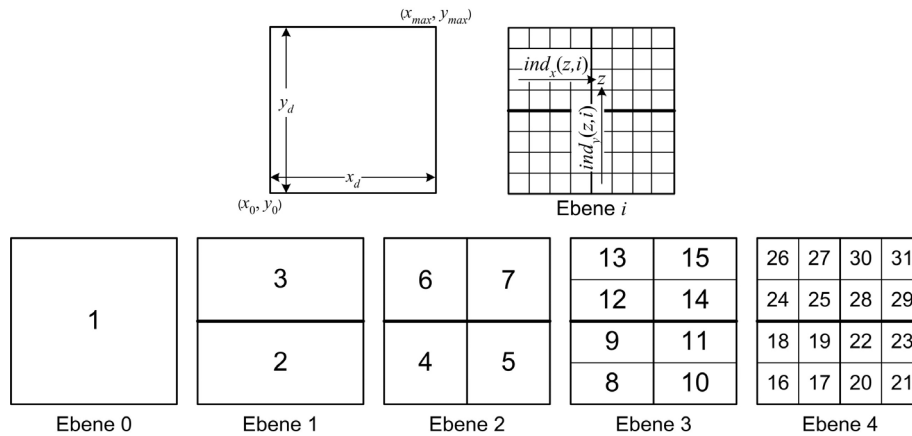


Abbildung 3: Idee des Extended Split Index

Diese Zuordnung ähnelt der *z-Kurve* [TH81], allerdings wird eine Kachel nicht in vier, sondern zwei Flächen zerlegt, zusätzlich werden die Kacheln fortlaufend über die Ebenen hinweg nummeriert. Einer vorgegebenen Geometrie werden später alle Kachelnummern verschiedener Ebenen zugeordnet, die diese Geometrie schneiden.

Häufig wird der Grad der Nachbarschaftserhaltung bei einer Index-Kurve betrachtet, also wie weit Kacheln mit geringem Abstand der Kachelnummern geometrisch entfernt sind. In diesem Zusammenhang wird häufig die Hilbert-Kurve diskutiert, die ein günstigeres Verhalten zeigt als die *z-Kurve*. Für unseren Ansatz ist jedoch diese Eigenschaft nebensächlich. Für den Extended Split Index ist wichtig, dass ein günstiger Zusammenhang zwischen Nummern von eingebetteten Kacheln auf *verschiedenen* Ebenen vorliegt.

Sei  $z$  eine Kachelnummer. Dann ist die Ebene  $i$  zu dieser Kachel  $i = \lfloor \log_2(z) \rfloor$  und die linke untere Kachel auf dieser Ebene hat die Nummer  $z_0 = 2^i$ . ( $\lfloor \cdot \rfloor$  rundet auf die nächstkleinere ganze Zahl ab.) Wir ordnen nun den Kachelnummern eine zweidimensionale Kachelkoordinate  $(ind_x(z, i), ind_y(z, i))$  aus  $(0 \dots 2^{\lfloor i/2 \rfloor} - 1, 0 \dots 2^{\lfloor (i+1)/2 \rfloor} - 1)$  zu (Abb. 3 oben). Hierzu benötigen wir eine Hilfsfunktion *interleave*: Sei  $n$  eine beliebige ganze Zahl mit

Binärrepräsentation  $(n_m n_{m-1} \dots n_3 n_2 n_1 n_0)_2$ . Weiter sei  $interleave_0(n) = (n_{m-1} \dots n_2 n_0)_2$  und  $interleave_1(n) = (n_m \dots n_3 n_1)_2$ . d.h. eine Zahl wird aus den geraden bzw. ungeraden Bits einer anderen Zahl gebildet. Dann können wir die Kachelkoordinaten wie folgt bilden:

$$ind_x(z, i) = interleave_{i \bmod 2}(z - 2^i), \quad ind_y(z, i) = interleave_{(i+1) \bmod 2}(z - 2^i) \quad (1)$$

Die Kantenlängen der Kacheln auf dieser Ebene sind dabei

$$rx_d(i) = \frac{x_d}{2^{\lfloor i/2 \rfloor}}, \quad ry_d(i) = \frac{y_d}{2^{\lfloor (i+1)/2 \rfloor}}. \quad (2)$$

Damit kann man einer ganzen Zahl  $z > 0$  eine Kachel mit den Eckpunkten

$$\begin{aligned} & (x_0 + ind_x(z, i) \cdot rx_d(i), y_0 + ind_y(z, i) \cdot ry_d(i)) \\ & (x_0 + (ind_x(z, i) + 1) \cdot rx_d(i), y_0 + (ind_y(z, i) + 1) \cdot ry_d(i)) \end{aligned} \quad (3)$$

zuordnen. Der Vorteil dieser Zuordnung: zu einer gegebenen Zahl  $z$ , ist die Menge *aller umgebenden Kacheln*

$$\left\{ \left\lfloor \frac{z}{2} \right\rfloor, \left\lfloor \frac{z}{4} \right\rfloor, \left\lfloor \frac{z}{8} \right\rfloor, \dots, 1 \right\} \quad (4)$$

sehr einfach zu berechnen. Umgekehrt werden *alle eingebetteten Kacheln* zu einer Kachel  $z$  durch die Menge

$$\{2 \cdot z, 2 \cdot z + 1\} \cup \{4 \cdot z, \dots, 4 \cdot z + 3\} \cup \{8 \cdot z, \dots, 8 \cdot z + 7\} \cup \dots \quad (5)$$

repräsentiert. Damit kann man einfach über den Vergleich von Kachelnummern erkennen, ob Geometrien potentiell überlappen.

Für die Benutzung des Indexes ist auch die Umkehrabbildung, also die Abbildung der Koordinaten auf eine Kachel erforderlich. Für einen Punkt  $(p_x, p_y)$  aus  $(x_0 \dots x_{max}, y_0 \dots y_{max})$  wird die Kachelnummer  $z$  auf der Ebene  $i$  wie folgt ermittelt:

$$z(p_x, p_y, i) = 2^i + \left\lfloor \frac{(p_x - x_0)}{rx_d(i)} \right\rfloor + \left\lfloor \frac{(p_y - y_0)}{ry_d(i)} \right\rfloor \cdot 2^{\lfloor i/2 \rfloor} \quad (6)$$

Die Idee des Indexes ist nun, einer beliebigen Geometrie aus diesem Bereich die größte Kachelnummer (also die kleinste Kachel) zuzuordnen, die diese Geometrie komplett enthält. Bei geometrischen Suchen ermittelt man diejenigen Kandidaten, die dieselbe Kachelnummer haben sowie die Nummern umgebener oder eingebetteter Kacheln (Formeln 4 und 5). Da eine Punktgeometrie durch beliebig kleine Kacheln umschlossen werden kann, wird eine maximale Ebenenzahl  $i_{max}$  definiert. Diese ist durch den Zahlenbereich festgelegt, der später durch die entsprechende Datenbankspalte zur Verfügung steht. Kann der entsprechende Datentyp Zahlen bis  $z_{max}$  speichern, so gilt

$$i_{max} = \lfloor \log_2(z_{max} + 1) \rfloor - 1 \quad (7)$$

Das führt zu einer weltweiten geometrischen Auflösung, wie in Tabelle 1 dargestellt.

Tabelle 1: Ebenenzahl in Abhängigkeit des Spaltentyps

Datentyp	$z_{\max}$	$i_{\max}$	Anzahl Kacheln in x-Richtung auf Ebene $i_{\max}$	$rx_d(i_{\max})$ am Äquator
INT2	127	6	8	5009 km
INT4	2147483647	30	32768	1,2 km
INT8	$9,22 \cdot 10^{18}$	62	2147483648	1,86 cm

Bei flächen- oder linienförmigen Geometrien bildet man die diagonalen Eckpunkte des umgebenden Rechtecks  $(p_{x0}, p_{y0}, p_{x\max}, p_{y\max})$  zunächst auf zwei Kachelnummern der höchsten darstellbaren Ebene ab. Sind diese unterschiedlich, so ermittelt man die *größte gemeinsame Kachelnummer*  $z_{rect}$  wie folgt:

$$z_{rect}(p_{x0}, p_{y0}, p_{x\max}, p_{y\max}) = \max(z_{ij} | z_{ij} = z(p_{x0}, p_{y0}, i) = z(p_{x\max}, p_{y\max}, j), i, j \in \{0, \dots, i_{\max}\}) \quad (8)$$

Zu bemerken ist, dass diese Funktion, auch wenn sie formal aufwändig aussieht, sehr einfach berechnet werden kann. Betrachtet man nämlich die Binärrepräsentation der Zahlen  $z(p_{x0}, p_{y0}, i_{\max})$  und  $z(p_{x\max}, p_{y\max}, i_{\max})$ , so entspricht die größte gemeinsame Kachelnummer  $z_{rect}$  dem gemeinsamen binären Präfix beider Nummern. Man kann daher  $z_{rect}$  einfach über binäre Schiebeoperationen berechnen.

Die größte gemeinsame Kachelnummer kann je nach Objektgröße eine bestimmte Ebene nicht überschreiten. So kann ein Objekt, das in x- oder y-Richtung einen Bereich von  $d_{obj}$  abdeckt, nur maximal bis zur Ebene

$$\min\left(2 \cdot \log_2\left(\frac{x_d}{d_{obj}}\right) + 1; 2 \cdot \log_2\left(\frac{y_d}{d_{obj}}\right)\right) \quad (9)$$

einsortiert werden. Das geht aber nur, wenn das entsprechende Objekt von der Lage exakt in eine Kachel passt, in der Regel wird solch ein Objekt aber in die nächstniedrigere Ebene eingeordnet. Kann man eine typische minimale Objektgröße definieren, so ist es sinnvoll, nicht den gesamten Bereich der verfügbaren Ebenenzahl auszunutzen. Unter Berücksichtigung der typischen minimalen Objektgröße ergibt sich folgende Formel für die maximale Ebenenzahl:

$$i_{\max} = \min\left(\lfloor \log_2(z_{\max} + 1) \rfloor - 1; 2 \cdot \log_2\left(\frac{x_d}{d_{obj}}\right); 2 \cdot \log_2\left(\frac{y_d}{d_{obj}}\right) - 1\right) \quad (10)$$

Tabelle 2 zeigt die maximal sinnvolle Ebenenzahl ausgewählter Objektgrößen, wenn die gesamte Erdoberfläche indiziert werden soll.

Tabelle 2: Ebenentiefe in Abhängigkeit der minimalen Objektgröße

$d_{obj}$	1 m	5 m	10 m	50 m	100 m	500 m	1 km
$i_{\max}$	51	46	44	40	38	33	31



Die Reduktion der maximalen Ebenenzahl kann nur erfolgen, wenn a priori-Wissen über den Datenbestand vorliegt, insbesondere welche Objektgrößen vorkommen. Liegt kein solches Wissen vor, verwendet man weiterhin die Formel 7.

Nach diesen Vorarbeiten wird der Index jetzt wie folgt verwendet:

- Zu jeder Geometrie wird in der Datenbank eine weitere Spalte für die Kachelnummer eingerichtet. Die Einrichtung geschieht für den Anwendungsentwickler unsichtbar.
- Für jede Einfüge- oder Änderungsoperation der Geometrie wird die Kachelnummer  $z$  gemäß der Formeln 6 und 8 berechnet und gespeichert.
- Werden Geometrien gelöscht, sind keine Maßnahmen notwendig.
- Bei Abfragen, die selbst Geometrien enthalten, wird die Kachelnummer der Abfragegeometrie ermittelt und die Abfrage um die Einschränkung der Index-Werte gemäß der Formeln 4 und 5 erweitert.

Die Einschränkung der Index-Werte der Abfrage hängt von der eigentlichen geometrischen Fragestellung ab. Bei Abfragen wie CONTAINS, OVERLAPS, WITHIN oder EQUALS werden zuerst alle Geometrien betrachtet, die eine Kachelnummer im erwarteten Bereich besitzen und danach über die Geometry-Engine exakt überprüft, ob die gesuchte geometrische Eigenschaft vorliegt. Eine Ausnahme bildet lediglich die Abfrage DISJOINT (alle Geometrien, die mit der Abfrage *keinen* gemeinsamen Punkt haben). Hier sind alle Geometrien, die eine Kachelnummer *außerhalb* des erwarteten Bereichs haben, ohne weitere Überprüfung ein Resultat. Da dadurch u.U. sehr große Ergebnismengen zurückgeliefert werden können, ist diese Operation mit Vorsicht anzuwenden.

Die Vorgehensweise bei der Index-Abfrage soll an einem Beispiel illustriert werden: Gesucht werden alle Städte in Deutschland mit einer Einwohnerzahl kleiner als 100 000. Der nichtgeometrische Anteil der Abfrage wird repräsentiert durch die Abfrage

```
SELECT * FROM CITIES WHERE INHABITANS<100000
```

Die Kachelnummer der durch die Grenzen von Deutschland repräsentierten Geometrie sei 55; sei weiter  $z_{\max}=511$ , d.h.  $i_{\max}=8$ . IND sei der Spaltenname für die Kachelnummer. Dann lautet die Abfrage insgesamt

```
SELECT * FROM CITIES WHERE INHABITANS<100000 AND
      (IND=55 OR IND=27 OR IND=13 OR
       IND=6 OR IND=3 OR IND=1
       OR IND BETWEEN 110 AND 117
       OR IND BETWEEN 220 AND 227
       OR IND BETWEEN 440 AND 447)
```

Jede Abfrage wird daher durch  $i_{\max}+1$  Bedingungen weiter eingeschränkt, die die Geometrie betreffen. Es ist dabei zu klären, ob die Länge eines SQL-Abfragestrings die durch das Datenbanksystem vorgegebene maximale Länge nicht überschreitet. Bei typischen Datenbanksystemen ist das allerdings unkritisch.

### 3.2 Weitere Einschränkung der Kandidatenmenge

Der bisher dargestellte Index hat noch ein Problem: Selbst kleine Geometrien können eine sehr niedrige Kachelnummer erhalten, wenn sie eine Trennlinie einer kleinen Ebenenzahl schneiden. So haben beispielsweise selbst sehr kleine Geo-Objekte, die auf dem Äquator liegen, die Kachelnummer 1. Solche Objekte werden immer in die Feinfilterung durch die Geometry-Engine einbezogen und führen zu einem konstanten Anteil jeder Abfrage. Weitere Geometrien, die auf den Trennlinien folgender Ebenen liegen, werden zwar nicht immer, aber in einem viel zu hohen Anteil überprüft.

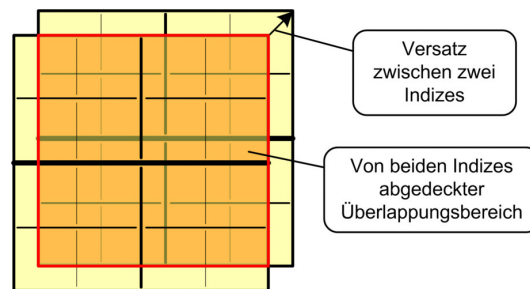


Abbildung 4: Die Idee versetzter Indizes

Der für den Extended Split Index gewählte Ausweg zu diesem Problem ist in Abb. 4 dargestellt. Es werden parallel zwei Kachelnummern berechnet. Die Index-Raster sind so gegeneinander verschoben, dass in der Regel nicht beide Indizes gleichzeitig eine kleine Kachelnummer produzieren. Positiv ausgedrückt: Man stellt sicher, dass mindestens eine der zwei Kachelnummern im Rahmen der Möglichkeit durch die Objektgröße hoch ist. Bei Abfragen werden die beiden Anteile, die sich auf die jeweiligen Kachelnummern beziehen durch AND verknüpft. Die höhere Kachelnummer schränkt die Resultatmenge dabei am größten ein.

Die Suche nach dem optimalen Versatz ist nicht trivial:

- Generell soll an den Stellen, wo einer der Indizes eine kleine Kachelnummer produziert, der andere eine große produzieren und umgekehrt.
- Objekte, die größer als der Versatz sind, können immer noch Trennlinien beider Indizes schneiden. Der Versatz soll daher groß genug sein, damit die Anzahl der Objekte, die beide Trennlinien schneiden, klein ist.
- Der Versatz darf aber nicht allzu groß sein, damit der Verschnitt an den Rändern der Koordinatensysteme nicht zu groß wird.

Zur Untersuchung der Fragestellung wurde eine Simulation auf der Basis realer Daten durchgeführt. Wichtig ist hierbei, dass die Daten eine typische Größenverteilung aufweisen: kleine Objekte kommen häufig vor, große Objekte selten, d.h. ab einer Mindestgröße liegt eine Exponentialverteilung vor [Roth05].

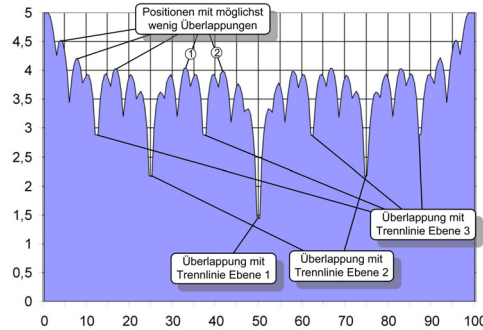


Abbildung 5: Durchschnittliche Index-Ebene bei Objekten an einer x-Position

Abb. 5 illustriert die Auswertung. Dargestellt ist die durchschnittliche Ebenenzahl in Abhängigkeit vom geometrischen Mittelpunkt eines Objektes. Zur einfacheren Darstellung wird nur eine Unterteilung in x-Richtung gezeigt. Die Position ist relativ angegeben (0:  $x_0$ , 100:  $x_{max}$ ). In Anlehnung an Formel 10 wird ein Index mit 5 Ebenen verwendet. Man kann folgendes beobachten:

- Da auf der x-Achse der Mittelpunkt des Objektes eingetragen wird und ein Objekt komplett auf dem adressierbaren Bereich liegen muss, erscheinen zum Rand hin nur noch kleine Objekte. Unterhalb einer gewissen Größe schneiden diese keine Trennlinie mehr und werden unter der Ebene 5 abgelegt.
- An den Positionen der Trennlinien nimmt die Ebenenzahl ab. Da sehr kleine Objekte aber auch nahe der Trennlinie mit hoher Ebenenzahl abgelegt werden können (wenn sie diese noch nicht schneiden), fällt der Durchschnittswert nicht auf 0.

Es gibt verschiedene lokale Maxima, die als Kandidaten für den Versatz interessant sind. Verschiebt man diese Maxima auf die Haupttrennlinie, haben entsprechende Objekte beim zweiten Index eine hohe Kachelnummer. Die Positionen (1) und (2) in der Abbildung sind dabei gute Kandidaten. Für die maximale Ebenenzahl  $i_{max}$  ist die relative Position (1)

$$\frac{1}{2} - \frac{1}{4} + \frac{1}{8} \dots \frac{1}{2^{i_{max}+1}} \approx \frac{1}{3} \quad \text{damit ist der Versatz } \frac{1}{2} - \frac{1}{3} = \frac{1}{6} \quad (11)$$

und für die relative Position (2)

$$\frac{1}{2} - \frac{1}{4} + \frac{1}{8} + \frac{1}{16} - \frac{1}{32} \dots \frac{1}{2^{i_{max}+1}} \approx \frac{5}{12} \quad \text{damit ist der Versatz } \frac{1}{2} - \frac{5}{12} = \frac{1}{12} \quad (12)$$

Abb. 6 zeigt das Resultat, wenn zwei Indizes mit Versatz verwendet werden. Für die vorgeschlagenen Versatz-Werte 1/6 und 1/12 ergeben sich nahezu konstante Ebenenwerte von 4 (allgemein  $i_{max}-1$ ). Daher sind diese Positionen auch durch diese Simulation als geeignete Kandidaten bestätigt worden. Die rechte Darstellung illustriert den Effekt, wenn der Versatz zu klein gewählt wird. Zu viele Objekte überlagern die Trennlinien beider Indizes, so dass im Ansatz wieder das typische Muster aus Abb. 5 entsteht.

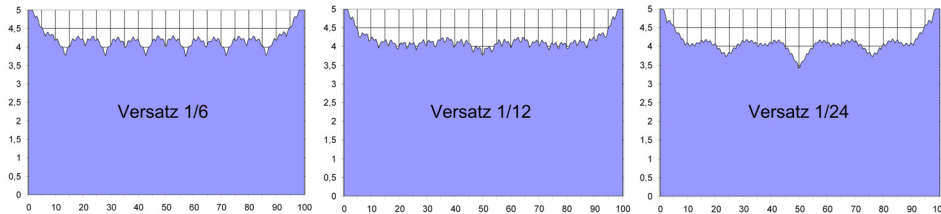


Abbildung 6: Durchschnittliche Index-Ebene bei zwei Indizes mit Versatz

Die Verwendung des Versatzes von 1/6 reduziert die verfügbare Fläche auf 69%, der Wert von 1/12 auf 84% der ursprünglichen Fläche. Um immer noch 100% der geforderten Fläche abzudecken, muss der Index 144% (bei 1/6), bzw. 119% (bei 1/12) der Fläche indizieren. Bei den verfügbaren Datentypen für die Index-Spalte stellen beide Verminderungen der zugehörigen räumlichen Auflösung keine nennenswerte Einschränkung dar.

### 3.3 Aufbau von Abfragen, Datenbankunabhängigkeit

Der räumliche Index wird vor dem Anwendungsentwickler vollständig verborgen. Darüber hinaus soll die Formulierung von geometrischen Abfragen über die objektorientierten Mittel der Anwendung möglich sein. Dies geschieht über die Komponente *Abfrage-Generierung* (Abb. 2) unter Zuhilfenahme der Komponente *Externe SQL-Generierung*. Als externe Komponente wird in der aktuellen Implementierung das Paket JaxME/SQLS verwendet. Dieses generiert über objektorientierte Aufrufe einen SQL-String. Für das GAO musste die Fähigkeit hinzugefügt werden, geometrische Anfragen separat zu behandeln und entsprechende Index-Abfragen in SQL zu generieren.

Als Beispiel: Gesucht werden alle Städte in Deutschland mit einer Einwohnerzahl kleiner 100 000. Die Anfrage wird folgendermaßen zusammengestellt:

```
select=dbman.getSelectStatement("CITIES"); // TABLE CITIES
condition=select.getWhere().createLT(); // CONDITION <
condition.addPart(select. // COL INHABITANS
    getSelectTableReference().newColumnReference("INHABITANS"));
condition.addPart(100000); // VALUE 10000
result=dbman.search(SearchType.WITHIN, GermanBorder,
    select); // INSIDE GERMANY
```

Die geometrische Bedingung wird über den letzten Aufruf ausgedrückt. In der Anwendung muss dabei die Variable `GermanBorder` die Geometrie der deutschen Grenze enthalten.

Obwohl Standard-SQL-Datenbanken prinzipiell gegeneinander austauschbar sind, ergeben sich noch einige Unterschiede, die durch die Komponente *DB-Zugriff* behandelt werden. Die wesentlichen Unterschiede sind:

- Der Spaltentyp für Geometrien: typischerweise steht hierfür der Typ `BLOB` zur Verfügung, aber auch `ByteA` (PostgreSQL), `Binary` (MS SQL Server) oder `Long-`

`VarBinary` (HSQLSDB). Für Datenbanken ohne binäre Datentypen kann die Geometrie auch als Zeichenkette in einer `VARCHAR`-Spalte abgelegt werden.

- Der Spaltentyp für Kachelnummern: in der Regel existiert für eine Ganzzahl mit 64 Bit der Typ `BIGINT`, Oracle hingegen verwendet hierfür `INTEGER`.
- Anhand welcher Parameter wird der Zugriff zur Datenbank hergestellt (z.B. Login, Kennwort).

Die Definition eines DB-Zugriffs umfasst in der Regel nicht mehr als 200 Zeilen Code.

### 3.4 Performance-Betrachtungen

Um den Ansatz zu bewerten, wurde eine Reihe von Performance-Messungen durchgeführt. Diese sollen einerseits den Vorteil des zweifachen versetzten Indexes gegenüber dem einfachen Index zeigen, andererseits die generelle Tauglichkeit des Ansatzes für geometrische Anfragen darlegen.

Konkrete Messungen zu verallgemeinern, ist generell schwierig, da neben dem reinen Algorithmus indirekt auch nicht kontrollierbare Anteile der Software (hier dem Datenbanksystem) sowie der Hardware (CPU, RAM) getestet werden. Außerdem hängen die gemessenen Zeiten signifikant von den Daten und den Anfragegeometrien ab. Die folgenden Messungen müssen daher zwar in diesem Sinne relativiert werden, zeigen aber dennoch einen deutlichen Trend.

Die Messungen wurden auf einem PC mit 2,49 GHz und 3 GB RAM unter Windows XP durchgeführt. Als Datenbanksystem wurde PostgreSQL in der Version 8.3 eingesetzt. Es wurden die insgesamt 226 497 Datensätze aus der Datei "Bayern" aus OpenStreetMap [OSM09] importiert (Stand Januar 2009). Die Datenbank benötigte für das einmalige Anlegen inklusive der Index-Werte 2,4 ms pro Datensatz. Das Ermitteln des doppelten Index-Wertes fällt dabei gegenüber dem einfachen Index-Wert nicht ins Gewicht.

Die erste Auswertung soll die Tiefe der Index-Ebenen zeigen – einmal bei einfachem Index, einmal bei zwei versetzten Indizes (Versatz 1/6). Bei den versetzten Indizes wird das Maximum der Ebenenzahl aus den einzelnen Index-Werten verwendet, da der höhere Wert die größere Einschränkung für die Kandidatenmenge verursacht. Abb. 7 zeigt das Ergebnis. Die durchschnittliche Index-Ebene liegt bei 32,8 für den einzelnen Index und 34,7 bei versetztem Index (Vergrößerung um ca. 2). Insbesondere traten beim versetzten Index keine Objekte mehr mit kleinen Index-Nummern auf.

Durch weitere Messungen wurden konkrete Laufzeiten bei Anfragen ermittelt (Tabelle 3). Es wurden nur geometrische Bedingungen formuliert, d.h. die Kandidatenmenge wurde nicht zusätzlich durch nicht-geometrische Bedingung verkleinert. Dargestellt sind die Zahl der Treffer, die Zahl der Kandidaten durch den räumlichen Index, die Zeit für die Datenbankanfrage (inklusive Deserialisierung) sowie die Zeit in der Geometry-Engine, die für die Feinfilterung verwendet wurde.

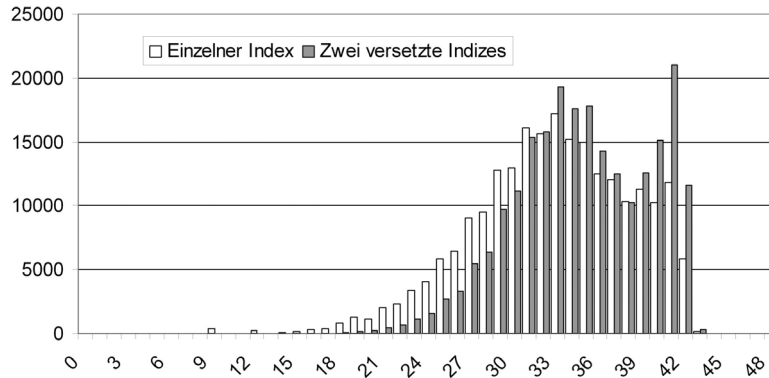


Abbildung 7: Verteilung der Index-Ebenen der Geodaten von Bayern

Tabelle 3: Ergebnisse der Performance-Messungen

Abfragetyp	Abfrage-Geometrie	Anzahl Treffer	Einzelner Index			Zwei versetzte Indizes		
			Anzahl Kandidaten	Zeit DB [ms]	Zeit Feinfilter [ms]	Anzahl Kandidaten	Zeit DB [ms]	Zeit Feinfilter [ms]
CONTAINS	POINT (49.4574 11.0827)	2	1410	781	0,219	89	46	0,011
CONTAINS	POINT (49.8000 11.7000)	0	706	421	0,000	22	12	0,000
OVERLAPS	POLYGON (48.9574 11.0827...)	2	831	375	0,235	21	11	0,006
CONTAINS	POLYGON (49.4500 12.9000...)	0	590	313	0,000	71	30	0,000
WITHIN	POLYGON (49.4520 11.0927...)	33	3178	1109	0,047	272	104	0,006
WITHIN	POLYGON (49.4574 11.0827...)	345	3178	1188	0,078	2859	969	0,055

Die Auswertungen ergeben, dass der versetzte Index die Kandidatenmenge signifikant einschränkt. Damit ist dieser Ansatz der einfachen Variante vorzuziehen. Für Laufzeiten ausschlaggebend sind die Zeiten in der Datenbank, also nicht die Zeit für die Feinfilterung. Der genaue Einfluss der Geometry-Engine ist allerdings sehr schwer zu fassen. Eine große Treffermenge (und damit Kandidatenmenge) erhöht zwar die Zeit im Feinfilter, allerdings sind verschiedene Optimierungen innerhalb der Geometry-Engine integriert, so dass die Zeiten für den Feinfilter-Vergleich pro Datensatz höchst unterschiedlich ausfallen können. Zusammenfassend ergibt die Auswertung, dass der versetzte Extended Split Index ein effizienter Ansatz mit geringen Einfügekosten und hinreichend kleiner Kandidatenmenge bei geometrischen Suchen ist. Die Laufzeiten sind akzeptabel und praxistauglich.

## 4 Zusammenfassung und Ausblick

Das HomeRun Add-on für räumliche Abfragen erlaubt ortsbezogenen Anwendungen und Diensten, räumliche Daten in Standard-Datenbanken zu verwalten. Das Add-on führt den neuartigen Extended Split Index ein, der Index-Werte isoliert für jede Datenzeile berechnet, somit auf die Reorganisation ganzer Index-Spalten verzichten kann. Der Datenzugriff erfolgt über den Objektraum der Anwendung.

Aktuell sind Abfragen nur anhand von Vergleichen mit einer gegebenen Geometrie möglich (z.B. welche Geo-Objekte liegen innerhalb eines bestimmten Polygons). Ein erstes zukünftiges Ziel ist, Abfragen über *abgeleitete* Eigenschaften der Geometrie (z.B. Flächeninhalt oder Durchmesser) zu realisieren. Eindimensionale abgeleitete Eigenschaften lassen sich leicht in weitere Spalten ablegen, müssen jedoch bei jeder Geometrie-Änderung korrigiert werden. Hier ist ein geeigneter Mechanismus zu entwickeln, damit nicht zu viele zusätzliche Spalten ständig mitgeführt werden müssen.

Langfristig sollen weitere Funktionen, die aus dem nicht-geometrischen Bereich bekannt sind, auch für geometrische Attribute übernommen werden. Die größte Herausforderung wird dabei sein, Geometrien aus mehreren Tabellen effizient in einer einzelnen Anfrage zu kombinieren (insb. das geometrische Join), ohne große Zwischenergebnisse im Objektraum der Anwendung generieren zu müssen.

## Literaturverzeichnis

- [Aq03] Aquino J.: JTS Topology Suite, Technical Specifications, Vivid Solutions, 2003
- [Bey09] Beyer M.: Konzeption und Realisierung eines Geospatial Add-Ons für SQL, Diplomarbeit, Ohm-Hochschule Nürnberg, März 2009
- [Bri07] Brinkhoff T.: Open-Source-Geodatenbanksysteme, Datenbank-Spektrum 22/2007, 37-43
- [FB74] Finkel R., Bentley J.L.: Quad Trees: A Data Structure for Retrieval on Composite Keys, Acta Informatica 4 (1): 1974, 1-9
- [Gut84] Guttman A.: R-Trees: A Dynamic Index Structure for Spatial Searching, Proc. of the 1984 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, June 1984, 47-57
- [Herr05] Herring J. (ed.): OpenGIS® Implementation Specification for Geographic information - Simple feature access - Part 1: Common architecture, OGC, 2005
- [Herr06] Herring J. (ed.): OpenGIS Implementation Specification for Geographic information - Simple feature access - Part 2: SQL option, OGC, 2006
- [Hib09] Hibernate Spatial Tutorial, 2009, <http://www.hibernate.org>
- [Mur08] Murray C.: Oracle Spatial Developer's Guide, Oracle, Aug. 2009
- [Neu09] Neufeld K.: PostGIS Manual, 2009
- [OSM09] OpenStreetMap excerpts for Europe, <http://download.geofabrik.de/osm/>
- [Roth05] Roth J.: A Decentralized Location Service Providing Semantic Locations, Informatik Bericht 323, Fernuniversität Hagen, Jan. 2005
- [Sto03] Stolze K.: SQL/MM Spatial: The Standard to Manage Spatial Data in Relational Database Systems, BTW 2003, Leipzig, Feb 2003
- [Sun09] MySQL 5.0 Reference Manual, Sun Microsystems, 2009
- [TH81] Tropf H., Herzog H.: Multidimensional Range Search in Dynamically Balanced Trees, Angewandte Informatik, 2/1981, 71-77