Anfragegetriebene Indizierung räumlicher Daten

Hannes Voigt, Steffen Preißler, Matthias Böhm, Wolfgang Lehner
Technische Universität Dresden
Lehrstuhl für Datenbanken
dbgroup@mail.inf.tu-dresden.de

Abstract: Mit der zunehmenden Verbreitung von GPS- und internetfähigen Smartphones werden ortsbezogene Informationsdienste immer beliebter. Zur Sicherung einer hohen Dienstqualität werden die zugrundeliegenden Ortsinformationen indiziert. Bekannte Indexstrukturen für räumliche Daten teilen diese gemäß ihrer Verteilung auf, wodurch alle Anfragen gleich behandelt werden. Möchte man häufige Anfrage durch eine genauere Indizierung besonders unterstützen, so muss sich die Aufteilung der Daten nicht an der Datenverteilung, sondern an der Anfrageverteilung orientieren. In diesem Papier stellen wir das QD-Grid vor, eine räumliche Indexstruktur, deren Indizierung sich inkrementell mit den gestellten Anfragen aufbaut. Zusätzlich präsentieren wir Evaluationsergebnisse.

1 Einleitung

Im Mobilfunkbereich ist die zunehmende Verbreitung leistungsstarker, internetfähiger Endgeräte, sogenannter Smartphones wie dem iPhone oder dem G1, zu beobachten. Getragen vom Erfolg der Smartphones, wächst auch die Nutzung ortsbezogener Informationsdienste stark. Denn ausgestattet mit Internetzugang und GPS-Empfänger sind Smartphones das ideale Endgerät für die Nutzung solcher Dienste. Eine häufige Art ortsbezogener Dienste ist die Umgebungsauskunft, bei der die Nutzer Informationen über die Umgebung ihres aktuellen Standortes erfragen, z.B. alle Sehenswürdigkeiten im Umkreis von 1000 Metern. Die Datenbestände solcher Dienste können erheblichen Umfang annehmen, wie das Beispiel des Dresdener Themenstadtplans[1] zeigen soll. Abbildung 1 zeigt eine Ausschnitt. Hier werden verschiedenste, ortsbezogene Informationen gebündelt, vom Baubestand über ÖPNV-Haltestellen und Kultureinrichtungen bis zu Abfallbehältern.

Die Nutzungscharakteristik solcher ortsbezogenen Informationsdienste begründet sich aus dem Nutzerverhalten im Allgemeinen und den Bewegungsmustern der Nutzer im Speziellen. Viele Nutzer bewegen sich oft auf ähnlichen Pfaden und nutzen an räumlich gleichen Stellen Umgebungsauskünfte. So ist die Anfragewahrscheinlichkeit z.B. in hoch frequentierten, touristischen Gebieten einer Stadt höher als in Wohngebieten, woraus sich eine starke Ungleichverteilung der Anfragewahrscheinlichkeit im Raum ergibt. Die räumliche Verteilung der Anfragewahrscheinlichkeit muss sich dabei nicht mit der räumlichen Verteilung der vom Informationsdienst erfassten Objekte decken. Als Beispiel soll hier die Auskunft über die nächste Straßenbahnhaltestelle dienen. Städte sind recht gleichmäßig

[1] http://themenstadtplan.dresden.de

Abbildung 1: Beispiel Themenstadtplan Dresden

mit Haltestellen abgedeckt. Jedoch ist die Anfragewahrscheinlichkeit an eine solche Auskunft in den touristischen Bereichen der Stadt deutlich höher als in bewohnten Gebieten, in denen die Straßenbahn vorwiegend von Ortskundigen genutzt wird.

Zur Beschleunigung der Verarbeitung von Umgebungsanfragen werden spezielle Indizierungstechniken für räumliche Daten eingesetzt. In unserem Szenario ist eine Indizierung wünschenswert, welche durch Anfragen getrieben wird, so dass Gebiete häufiger Anfragen feingranularer indiziert und Änderungen nur bei Interesse in die Indizierung eingepflegt werden. Dadurch lässt sich (1) in Gebieten mit häufigen Anfragen die höchste Dienstqualität erbringen, (2) der Ressourcenaufwand häufig gestellter Anfragen reduzieren und (3) der Ressourcenaufwand von Änderungsoperationen sehr gering halten. Alle bekannten Indizierungstechniken für Umgebungsanfragen auf räumlichen Daten werden ausschließlich durch Datenänderungen gepflegt und orientieren sich dabei nur an der räumlichen Verteilung der indizierten Objekte.

In diesem Papier stellen wir das QD-Grid vor, eine räumliche Indexstruktur zur Unterstützung von Umgebungsanfragen, deren Indizierung sich inkrementell mit den gestellten Anfragen aufbaut. Das QD-Grid basiert auf dem herkömmlichen Grid-File [NHS84]. Grid-Files untergliedern den gesamten zu indizierenden Raum in rechteckige Bereiche ähnlicher Anzahl an Objekten, orientieren sich also an der Objektverteilung. Das heißt, Regionen hoher Objektdichte werden feiner indiziert, unabhängig von der Anfragehäufigkeit. Die Anpassung der Untergliederung erfolgt beim Einfügen, Ändern und Löschen, also unabhängig davon, ob die geänderten Objekte bei Anfragen überhaupt von Interesse sind. Das hier vorgestellte QD-Grid unterteilt den zu indizierenden Raum in Regionen häufiger Anfragen feiner. Regionen geringer Anfragehäufigkeit werden grob unterteilt. Das QD-

Grid wird inkrementell durch die gestellten Anfragen verfeinert und gewartet. Eine Anfrage verfeinert dabei immer nur die Daten, die sie zur Beantwortung der Anfrage ohnehin lesen muss.

Der Rest des Papiers ist wie folgt aufgebaut. Zunächst wird die grundlegende Struktur des QD-Grids und die inkrementelle Verfeinerung mithilfe von Anfragen beschrieben (Abschnitt 2). Anschließend wird die Realisierung von Änderungsoperationen diskutiert (Abschnitt 3). Es folgt eine Vorstellung der durchgeführten experimentellen Evaluierung (Abschnitt 4). Das Papier endet mit einem Überblick über verwandte Arbeiten (Abschnitt 5) und einer Zusammenfassung (Abschnitt 6).

2 Aufbau und Anfragen

Das QD-Grid orientiert sich am Grid-File. Das Grid-File untergliedert die Menge zu indizierender Objekte durch Grenzlinien im Raum, welche parallel zu den Dimensionsachsen verlaufen. Diese Grenzlinien orientieren sich an der Objektverteilung und entstehen mit dem Einfügen von Objekten. Die Grundidee des QD-Grid ist es, Grenzlinien mit den gestellten Anfragen anzulegen. Beim Auswerten von Anfragen entsteht Wissen über die räumliche Position der zu indizierenden Objekte, das heißt ob sie links oder rechts einer bestimmten Grenzlinie liegen. Diesen Wissen wird durch Anlegen neuer Grenzlinien in das QD-Grid eingepflegt und das QD-Grid so nach und nach aufgebaut.

Zunächst wird der strukturelle Aufbau des QD-Grids skizziert. Anschließend wird beschrieben, wie das QD-Grid sukzessive durch die gestellten Anfragen verfeinert wird.

2.1 Struktur eines QD-Grids

Indiziert wird eine Menge von Objekten o in einem Raum. Aus Gründen besserer Verständlichkeit und ohne Beschränkung der Allgemeinheit betrachten wir hier diesen Raum als zwei-dimensional. Der zu indizierende Raum wird durch ein Gitter achsenparalleler Grenzlinien in Zellen aufgeteilt. Das Gitter wird durch zwei Skalen X und Y von n bzw. m Dimensionswerten beschrieben. Die Skale $X = [x_0, \ldots, x_{n-1}]$ enthält die Schnittstellen aller zur y-Achse parallelen Grenzlinien mit der x-Achse in aufsteigender Reihenfolge. Die Skale $Y = [y_0, \ldots, y_{m-1}]$ enthält analog die Schnittstellen aller zur x-Achse parallelen Grenzlinien mit der y-Achse. Jeden Zelle $c_{i,j}$ wird durch ihre Position im Gitter (i-te Spalte, j-te Zeile) identifiziert und verweist auf einen Container von indizierten Objekten. In einem zwei-dimensionalen Verzeichnis D sind Verweise für jede Zelle hinterlegt, wobei $D[i, j]$ den Verweis auf den Container der Zelle $c_{i,j}$ enthält. Mehrere Zellen können auf den gleichen Container verweisen. Ein Container beinhaltet immer genau die Objekte, welche in den Zellen liegen, die auf den Container verweisen. Beim herkömmlichen Grid-File ist die Größe eines Containers auf die Größe einer Seite begrenzt und das Überlaufen eines Containers führt zum Einfügen neuer Grenzlinien und dem damit verbundenen Teilen des Containers. Im QD-Grid werden die Grenzlinien durch die darauf

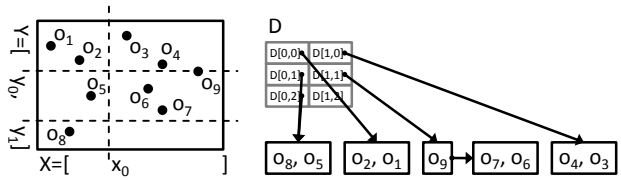


Abbildung 2: Struktureller Aufbau eines QD-Grids

gestellten Anfragen bestimmt, so dass die Nutzung einer fixen Containergröße nicht praktikabel ist. Daher sind Container je nach Größe auf mehrere verkettete Seiten verteilt. Die Objekte liegen dabei in umgekehrter Reihenfolge ihres Einfügens in den Seiten und jede neue Seite wird der Kette vorangestellt. Das bedeutet, das zuerst in den Container eingefügte Objekt liegt am Ende der letzten Seite der Kette. Um auf die Kette zu verweisen, wird die erste, d.h. die jüngste Seite der Kette, im Verzeichnis D referenziert.

Für jede Zelle verwaltet des QD-Grid den Median alle Objekte der Zelle. Dazu wird im Kopf jeder Seite eines Containers die Anzahl enthaltener Objekte n und der Median m gespeichert. Die Werte n_p und m_p einer Seite p beziehen sich dabei stets auf die Objekte in der Seite selbst und in allen älteren, p nachgestellten Seiten der Kette. Die jüngste Seite eines Containers am Anfang der Kette beinhalten somit die Objektanzahl n und den Median m für den gesamten Container der referenzierenden Zelle. Der Median einer Zelle lässt sich so sehr einfach auslesen und beim Einfügen von Objekten warten.

Abbildung 2 zeigt den strukturellen Aufbau eines QD-Grids am Beispiel von neun indizierten Objekten. Der Datenraum, in dem die Objekte liegen, ist durch das Gitter des QD-Grids in sechs Zellen aufgeteilt. Getrennt werden die Zellen durch eine y-parallele Grenzlinie x_0 und zwei x-parallele Grenzlinie y_0 und y_1. Das Verzeichnis D enthält für jede Zelle einen Seitenverweis. Der Verweis für Zelle $c_{1,2}$ ist auf NULL gesetzt, da in dieser Zelle keine Objekte liegen. Zelle $c_{0,1}$ und $c_{0,2}$ verweisen auf die gleiche Seite, welche entsprechend das Objekt o_5 aus Zelle $c_{0,1}$ sowie das Objekt o_8 aus Zelle $c_{0,2}$ enthält. In Zelle $c_{1,1}$ liegen die drei Objekte o_6, o_7 und o_9. Da im Beispiel eine Seite nur für zwei Objekte Platz bietet, verweist $c_{1,1}$ auf eine Kette von zwei Seiten. Die Einfügereihenfolge wird durch die Indizes der Objekte dargestellt. Es ist gut sichtbar, dass früher eingefügte Objekte in den Seiten und Seitenketten am Ende, später eingefügte Objekte am Anfang stehen. Objektanzahl und Mediane der Zellen sind im Beispiel nicht dargestellt.

2.2 Aufbau des QD-Grids durch Anfragen

Beim QD-Grid wird das Gitter nicht beim Einfügen von Objekten aufgebaut, sondern beim Beantworten von Anfragen. Entsprechend besteht nach dem Anlegen des QD-Grids das Gitter aus nur einer großen Zelle, das heißt die Skalen X und Y sind leer und alle Objekte befinden sich im Datencontainer der Zelle $c_{0,0}$. Mit jeder Anfrage wird das Gitter schrittweise verfeinert. Das Prinzip der Verfeinerung wird zunächst für Punktanfragen beschrieben und anschließend auf Umgebungsanfragen ausgeweitet.

2.2.1 Punktanfragen

Wird nach allen Objekten am Punkt q gefragt, so ermittelt die Anfrageroutine zunächst aus den Listen X und Y die Zelle c_q, in welcher der angefragte Punkte q liegt. Die Funktion $pos(X, q.x)$ liefert dabei die Position i der nächsten y-parallelen Grenzlinie der Skale X, also das kleinste i mit $q.x < x_i$ bzw. n wenn gilt $q.x \geq x_{n-1}$. Analog liefert $pos(Y, q.y)$ die Position j der nächsten x-parallelen Grenzlinie der Skale Y. Über $D[i, j]$ kann nun auf genau die Objekte zugegriffen werden, die in Zelle c_q liegen. Mittels Durchsuchen aller Objekte in c_q findet die Anfrageroutine schließlich alle Objekte am gewünschten Punkt q.

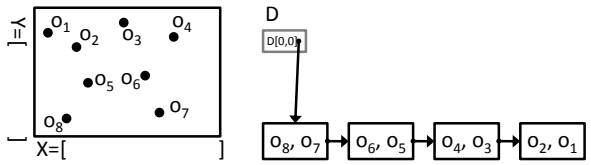
Dieses grundlegende Vorgehen beim Beantworten von Punktanfragen bleibt unverändert. Zusätzlich wird jedoch die Zelle, in welcher der angefragte Punkt liegt, neu aufgeteilt, wobei der Median der Zelle die Grenzlinien für die Aufteilung vorgibt. Algorithmus 1 stellt das Vorgehen im Detail dar.

Wird eine Punktanfrage auf den Punkt q an das QD-Grid gestellt, so wird mit Hilfe der Skalen zunächst die Zelle c_q, in der sich der angefragte Punkt befindet, ermittelt (Zeile 2). Beim Durchsuchen der Zelle c_q werden die Objekte der Zelle c_q gemäß der Anfrage neu aufgeteilt und so das QD-Grid verfeinert. Der Algorithmus ließt dazu den für Zelle c_q gespeicherten Median m aus der ersten Seite des von der Zelle referenzierten Datencontainers (Zeile 6). Auf Basis von m fügt Algorithmus in die Skalen X und Y an der Position $pos(X, m.x)$ beziehungsweise $pos(Y, m.y)$ die neue Grenzlinie $m.x$ beziehungsweise $m.y$ ein (Zeile 7 und 8). Es entstehen so feingranularere Zellen an dem angefragten Bereich. Anschließend dupliziert der Algorithmus im Verzeichnis D die Spalte $pos(X, m.x)$ sowie die Zeile $pos(Y, m.y)$, so dass D um eine Spalte und eine Zeile wächst (Zeile 9 und 10). Nun werden nacheinander die Seiten des Datencontainers der Zelle c_q gelesen (Zeile 11–33) und die enthaltenen Objekte gemäß der neu eingeführten Grenzlinien auf neue Datencontainer aufgeteilt (Zeile 13–25). Gleichzeitig werden die Objekte nach möglichen Ergebnissen durchsucht (Zeile 26). Die Schreibroutine übernimmt dabei das rückwärtige Auffüllen der Seiten, die Wartung der Mediane, die Allokation neuer Seiten und realisiert die Seitenverkettung, so dass die zuletzt beschriebene Seite die erste der Kette ist. Sind die betroffenen Objekte neu aufgeteilt, werden die neuen Container ins Verzeichnis D eingetragen (Zeile 34–37). Abschließend wird die Ergebnismenge R zurückgegeben (Zeile 38).

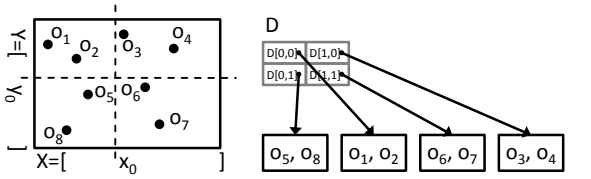
Abbildung 3 zeigt den Ablauf an einem Beispiel mit zwei Anfragen. Initial, dargestellt in Abbildung 3(a), besteht das QD-Grid aus einer einzigen großen Zelle $c_{0,0}$ und verweist genau auf eine Kette von Seiten, die alle indizierten Objekte beinhalten. Die Seitengröße im Beispiel ist auf zwei Objekte pro Seite beschränkt. Die Skalen X und Y sind leer. Nun wird einen Punktanfrage auf das Objekt o_6 gestellt. Abbildung 3(b) zeigt das nach dieser Anfrage verfeinerte QD-Grid. Beim Beantworten der Anfrage wird zunächst der Median $m_{0,0}$ der Zelle $c_{0,0}$ aus der ersten Seite der zugehörigen Seitenkette gelesen. Anschließend werden die Skalen X und Y um die Grenzlinie $x_0 = m_{0,0}.x$ beziehungsweise $y_0 = m_{0,0}.y$ erweitert. Die alte Zelle $c_{0,0}$ in der sich o_6 befindet wird wie oben beschrieben in die vier Gruppen $\{o_5, o_8\}$, $\{o_1, o_2\}$, $\{o_6, o_7\}$ und $\{o_3, o_4\}$ neu aufgeteilt. Das Verzeichnis D enthält nun vier Verweise.

Algorithmus 1 Punktanfrage

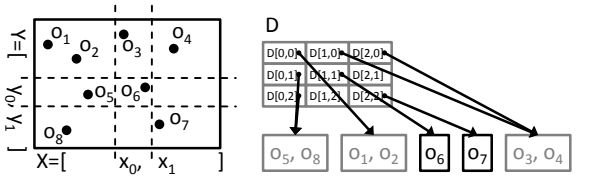
```
1:  procedure PointQuery(q)                                   ▷ q: Anfragepunkt
2:      i ← pos(X, q.x), j ← pos(Y, q.y)                      ▷ Zelle c_q, in der q liegt, bestimmen
3:      R ← ∅                                                 ▷ Ergebnismenge initialisieren
4:      z ← D[i, j]                                           ▷ Lesezeiger auf erste Seite von Zelle c_q setzen
5:      z_{0,0} ← z_{0,1} ← z_{1,0} ← z_{1,1} ← NULL          ▷ Schreibzeiger initialisieren
6:      m ←ReadMedian(z)                                      ▷ Zell-Median lesen
7:      Insert(X, i, m.x)   ▷ m.x als neue Grenzlinie an Stelle i in Skale X einfügen
8:      Insert(Y, j, m.y)   ▷ m.y als neue Grenzlinie an Stelle j in Skale Y einfügen
9:      DuplicateCol(D, i)                                    ▷ Duplizieren der i-ten Spalte im Verzeichnis D
10:     DuplicateRow(D, j)                                    ▷ Duplizieren der j-ten Zeile im Verzeichnis D
11:     while z ≠ NULL do                                     ▷ Durchsuchen und Neuaufteilen
12:         while o ←ReadNextObject(z) do                     ▷ Über Objekte der Seite laufen
13:             if o.x < m.x then                             ▷ Objekt o in neue Zelle sortieren
14:                 if o.y < m.y then
15:                     z_{0,0} ←Write(z_{0,0}, o)
16:                 else
17:                     z_{0,1} ←Write(z_{0,1}, o)
18:                 end if
19:             else
20:                 if o.y < m.y then
21:                     z_{1,0} ←Write(z_{1,0}, o)
22:                 else
23:                     z_{1,1} ←Write(z_{1,1}, o)
24:                 end if
25:             end if
26:             if o.x = q.x ∧ o.y = q.y then                 ▷ Wenn o gleich Anfragepunkt q
27:                 R ← R ∪ {o}                               ▷ o der Ergebnismenge R hinzufügen
28:             end if
29:         end while
30:         z′ ←GetNext(z)                                    ▷ Zeiger auf nächste Seite im Container ermitteln
31:         Delete(z)                                         ▷ Fertig gelesene Seite löschen
32:         z ← z′                                            ▷ Lesezeiger auf nächste Seite setzen
33:     end while
34:     D[i, j] ← z_{0,0}                                     ▷ Neue Container ins Verzeichnis eintagen
35:     D[i, j + 1] ← z_{0,1}
36:     D[i + 1, j] ← z_{1,0}
37:     D[i + 1, j + 1] ← z_{1,1}
38:     return R
39: end procedure
```



(a) Initialer Zustand des QD-Grids



(b) Zustand des QD-Grids nach der ersten Anfrage des Objekte o_6



(c) Zustand des QD-Grids nach der zweiten Anfrage des Objekte o_2

Abbildung 3: Verfeinerung des QD-Grids durch Punktanfragen

Eine zweite Anfrage, nun auf das Objekt o_7, verfeinert das QD-Grid weiter. Abbildung 3(c) zeigt das QD-Grid nach Beantwortung dieser Anfrage. Das Objekt o_7 liegt vor Beantwortung der Anfrage in der Zelle $c_{1,1}$, entsprechend wird der Median $m_{1,1}$ gelesen und die Grenzlinien $x_1 = m_{1,1}.x$ und $y_1 = m_{1,1}.y$ den Skalen hinzugefügt. Neu aufgeteilt wird nun jedoch nicht mehr die gesamte Menge von Objekten sondern nur die Objekte der Zelle, in der sich das angefragte Objekt o_7 befindet, im Beispiel die alte Zelle $c_{1,1}$. Durch die neuen Grenzlinien wird die alten Zelle $c_{1,1}$ zerschnitten und ihre Objekte o_6 und o_7 neu organisiert. Es ergeben sich vier neuen Zellen mit zugehörigen Objekt: $c_{1,1} \rightarrow \{o_6\}$, $c_{2,1} \rightarrow \emptyset$, $c_{1,2} \rightarrow \emptyset$, $c_{2,2} \rightarrow \{o_7\}$. Verweise auf die neu geschriebenen Seiten werden entsprechend ins Verzeichnis D eingetragen. Die neue Grenzlinie x_1 durchschneidet auch die alte Zelle $c_{1,0}$ und teilt sie in die neuen Zellen $c_{1,0}$ und $c_{2,0}$ auf. Jedoch werden die zugehörigen Objekt o_3 und o_4 nicht neu organisiert, da die Zelle nicht Ziel der Anfrage ist. Stattdessen verweisen beide neuen Zellen $c_{1,0}$ und $c_{2,0}$ auf die gleiche, unveränderte Seite wie die alte Zelle $c_{1,0}$. Analog wird mit der alten Zelle $c_{0,1}$ verfahren. Alle unveränderten Seiten sind grau dargestellt. Man sieht an diesem Beispiel gut, das die Anfrageroutine nur die Objekte neu verteilt, die sie zum Beantworten der Anfrage sowieso lesen muss. Das bei der Anfrageauswertung gewonnenes Wissen über die Objekte wird in die Objektorganisation eingebracht und so die Beantwortung späterer ähnlicher Anfragen vereinfacht.

2.2.2 Umgebungsanfragen

Eine Umgebungsanfrage (q, r) sucht nach allen Objekten o, die weniger als r Längeneinheiten Punkt q entfernt liegen, für die also gilt $dist(o, q) < r$. Um mit Hilfe eines QD-Grids Umgebungsanfragen zu beantworten, abstrahiert man die Kreisumgebung zu einem Rechteck, welches von den zwei Punkten $r_1 = (q.x - r, q.y - r)$ und $r_2 = (q.x + r, q.y + r)$ aufgespannt wird. Aus beiden Skalen X und Y ermittelt man alle Zellen, die von diesem Rechteck teilweise oder ganz abgedeckt werden. Die Funktion $range(X, q.x - r, q.x + r)$ liefert dabei die Menge I von Positionen i aller y-paralleler Grenzlinien der Skale X, die zwischen $q.x - r$ und $q.x + r$ liegen sowie die Position der nächsten y-parallelen Grenzlinie nach $q.x + r$. Also ist $I = \{i | q.x - r < x_i < q.x + r\} \cup \{pos(X, q.x + r)\}$. Analog liefert $range(Y, q.y - r, q.y + r)$ die Menge J mit $J = \{j | q.y - r < y_j < q.y + r\} \cup \{pos(Y, q.y + r)\}$. Das Kreuzprodukt $K = I \times J$ umfasst nun alle Zellen, für deren Objekte o das Kriterium $dist(o, q) < r$ zu prüfen ist.

Das Prinzip der Verfeinerung des QD-Grids durch Punktanfragen lässt sich einfach auf Umgebungsanfragen erweitern. Analog zu Punktanfrage teilt die Anfragebearbeitung auch bei der Umgebungsanfrage von der Anfrage betroffene Zellen neu auf. Konkret werden alle Zellen, die vom Rand des Anfragerechteckes durchschnitten werden, reorganisiert. Dafür wird für alle Zellen einer Seite des Anfragerechteckes aus den Zellmedianen und den dazugehörigen Zellkardinalität ein gemeinsamer Median bestimmt, nach des Wert die Zellen aufgeteilt werden. Alle y-parallel durchschnittenen Zellen werden dabei in der x-Dimension neugeaufteilt, alle x-parallel durchschnittenen Zellen entsprechend in der y-Dimension. Je nach Lage der angefragten Umgebung und des bestehenden Gitters sind dann im Gegensatz zur Punktanfrage mehrere Zellen zu reorganisieren. In jedem Fall werden aber nur die Zellen neu organisiert, die durch das Anfragerechteck teilweise abgedeckt werden, die also zur Beantwortung der Anfrage sowieso gelesen werden müssen.

3 Änderungsoperationen

Bisher wurden nur Anfrageoperationen des QD-Grids diskutiert. Dieser Abschnitt beschäftigt sich mit den Änderungsoperationen Einfügen und Löschen von Objekten. Das Verändern eines Objektes kann leicht durch Löschen der alten Objektversion und anschließendem Einfügen der neuen Objektversion realisiert werden und wird daher hier nicht weiter diskutiert.

3.1 Einfügen

Neue Objekte werden wie beim herkömmlichen Grid-File ins QD-Grid einsortiert, jedoch führt das Einfügen eines neuen Objektes nie zur Reorganisation. Algorithmus 2 zeigt das Vorgehen in Pseudocode. Ist die Zelle $c_{i,j}$, in der das einzufügende Objekt o liegt, mit Hilfe der Skalen identifiziert (Zeile 2), wird o in die erste Seite des Datencontainers der

Zelle $c_{i,j}$ geschrieben (Zeile 3). Sollte die erste Seite vollständig gefüllt und kann diese kein weiteres Objekt aufnehmen, stellt ihr die Schreibroutine eine neue erste Seite voran, fügt in diese das Objekt o ein und gibt einen Zeiger auf die neue Seite zurück. Die Einfügeroutine ändert in diesem Fall zudem den Verweis $D[i, j]$ auf die neue Seite (Zeile 4–6). Bei jedem Schreiben eines Objektes aktualisiert die Schreibroutine zusätzlich die Zellkardinalität sowie den Zellmedian.

Algorithmus 2 Einfügen

```
1: procedure INSERT(o)                          ▷ o: Einzufügendes Objekt
2:     i ← POS(X, o.x), j ← POS(Y, o.y)         ▷ Zelle c_o, in der o liegt, bestimmen
3:     z ←WRITE(D[i, j], o)                     ▷ o auf erste Seite von Zelle c_o schreiben
4:     if z ≠ D[i, j] then   ▷ Lieferte Schreibroutine neuen Erste-Seite-Zeiger zurück?
5:         D[i, j] ← z                          ▷ neuen Zeiger ins Verzeichnis eintragen
6:     end if
7: end procedure
```

3.2 Löschen

Analog zum Einfügen wird auch beim Löschen von Objekten das QD-Grid nicht reorganisiert. Stattdessen werden die betroffenen Objekte nur als gelöscht markiert. Führt eine Anfrage später eine Verfeinerung auf der betroffenen Zelle durch, werden die markierten Objekte auch physisch aus der Datenbasis entfernt. Durch dieses Vorgehen wird das beim Entfernen von Objekten nötige Konsolidieren dünnbesetzter Seiten aufgeschoben, bis die betroffenen Seiten ohnehin neugeschrieben werden. Es entsteht damit nur ein geringer Mehraufwand.

Zu löschende Objekte werden nicht direkt markiert. Bei einer direkten Markierung müssten alle Seiten, die ein zu löschendes Objekt enthalten können, gelesen werden. Zudem müsste jede Seite, die ein zu löschendes Objekt enthält, neu geschrieben werden, so dass der Aufwand dem einer physischen Löschung entspricht. Unser QD-Grid verwendet eine indirekte Markierung über ein sogenanntes Löschobjekt. Sollen alle Objekte am Punkt l gelöscht werden (Punktlöschung), so erzeugt die Löschroutine für diesen Punkt ein spezielles Löschobjekt o_l mit $o.x = l.x$ und $o.y = l.y$ und fügt es mittels der Einfügeroutine ins QD-Grid ein. Eine Löschoperation entspricht damit der Einfügeoperation. Die eigentliche Löscharbeit wird durch entsprechende Anfragen durchgeführt, welche den betroffenen Bereich reorganisieren.

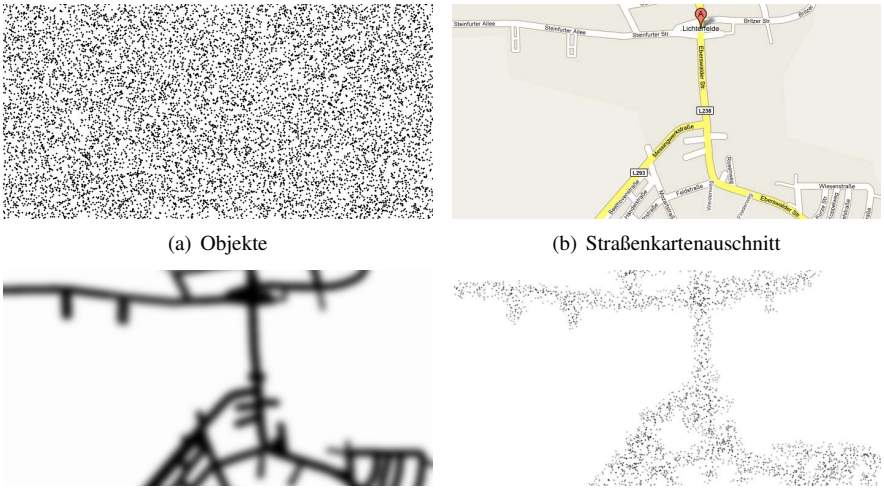
Algorithmus 3 zeigt den entsprechend erweiterten Pseudocode für Punktanfragen. Zur besseren Übersichtlichkeit sind unveränderte Teile ausgelassen. Die Anfrageroutine verwaltet zusätzlich ein Löschpredikat f, welchem zu einem Zeitpunkt des Lesen genau die Objekte genügen, für die zuvor Löschobjekte gelesen wurden. Zunächst initiiert die Anfrageroutine das Löschpredikate mit $false$, da initial keine Objekt zu löschen sind (Zeile 3). Anschließend werden wie bereits beschrieben, die Objekte der betroffenen Zelle gelesen. Ließt die Anfrageroutine ein Löschobjekt o_l (Zeile 6), erzeugt sie ein entsprechendes Filter-

kriterium f_l (Zeile 7) und fügt es ODER-verknüpft dem Löschpredikat f hinzu (Zeile 8). Weitergehend werden Löschobjekte nicht behandelt. Jedes Datenobjekt o, welches gelesen wird, prüft die Anfrageroutine nun gegen das aktuelle Löschpredikat f. Ist f für o erfüllt, so wird o nicht weiter behandelt (Zeile 11). Entsprechend wird das als gelöscht markierte Objekt o nicht mit reorganisiert und mit dem Freigeben der Seite, die o enthält, schließlich physisch gelöscht. Das Löschen von Objekten verursacht bei der Anfrageverarbeitung durch das Lesen von Löschobjekten nur einen geringen Mehraufwand.

Algorithmus 3 Punktanfrage mit Auswertung von Löschobjekten

```
1:  procedure POINTQUERY(q)                          ▷ q: Anfragepunkt
2:      ⋮
3:      f ← false                                    ▷ Löschpredikat f initieren
4:      while z ≠ NULL do                            ▷ Durchsuchen und Neuaufteilen
5:          while o ←READNEXTOBJECT(z) do            ▷ Über Objekte der Seite laufen
6:              if ISDELETEOBJEKT(o) then            ▷ Wenn o ein Löschobjekt ist
7:                  f_l ← GETCRITERION(o)             ▷ Löschkriterium aus o erzeugen
8:                  f ← f ∨ f_l                      ▷ dem Löschpredikat f hinzufügen
9:                  continue                         ▷ Löschobjekt nicht weiter behandeln
10:             end if
11:             if SATISFIES(o, f) then              ▷ Wenn o dem Löschpredikat f genügt
12:                 continue                         ▷ o nicht weiter behandeln
13:             end if
14:             ⋮   ▷ Objekt o in neue Zelle sortieren und gegen Anfrage prüfen
15:         end while
16:         ⋮
17:     end while
18:     ⋮
19: end procedure
```

Unter der Voraussetzung, dass alle Operationen auf dem QD-Grid transaktional serialisiert ausgeführt werden, ist das beschriebene Löschverfahren korrekt. Alle Objekte, das heißt Daten- als auch Löschobjekte, eines Containers werden stets in der Reihenfolge gelesen, in der sie eingefügt wurden. Dies wird dadurch sichergestellt, dass Datencontainer immer von hinten nach vorn befüllt und von vorn nach hinten gelesen werden. Wird zum Zeitpunkt t_i ein Löschobjekt o_l eingefügt, so wird es allen anderen Objekten in der Seitenkette vorangestellt. Alle Objekte hinter o_l sind folglich ältere Objekte, die zum Zeitpunkt t_i bereits sichtbar sind. Erfüllen sie das von o_l repräsentierte Löschkriterium, ist ihre Löschung demnach korrekt. Wird zum Zeitpunkt t_{i+1}, also später, ein neues Objekt o_n eingefügt, so wird es allen anderen Objekten in der Seitenkette inklusive dem Löschobjekt o_l vorangestellt. Das Objekt o_n wird beim Einlesen der Seite entsprechend vor o_l gelesen und gegen f geprüft bevor das von o_l repräsentierte Löschkriterium f hinzugefügt wird. Folglich kann ein Löschobjekt o_l nicht zur Löschung eines Objektes o_n führen, welches nach der Löschoperation eingefügt wurde.



(a) Objekte (b) Straßenkartenauschnitt

(c) Graufstufenbild (d) Punktanfragen

Abbildung 4: Szenario

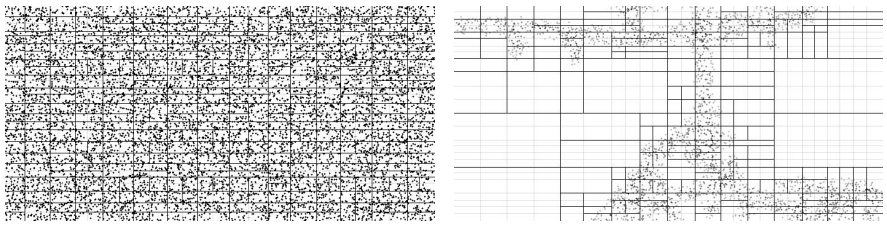
Das beschriebene Verfahren lässt sich neben der Punktlöschung auch für andere Löschoperationen, wie der Objektlöschung oder der Bereichslöschung verwenden. Bei der Objektlöschung ist ein Objekt o gegeben, welches aus dem QD-Grid gelöscht werden soll. Hierzu muss lediglich im zugehörigen Löschobjekt ein entsprechend nur auf o positiv auswertendes Löschkriterium hinterlegt werden. Bei der Bereichslöschung sind alle Objekte eines gegebenen Bereichs zu löschen. Dafür muss das Löschobjekt in alle Zellen eingetragen werden, die vom zu löschenden Bereich teilweise abgedeckt werden.

4 Experimente

Zur Evaluierung des Ansatzes wurden Experimenten durchgeführt, die in diesem Abschnitt vorgestellt werden. Alle Experimente sind in einem synthetischen Szenario angesiedelt, dargestellt in Abbildung 4. Zu indizieren waren 10000 Objekte, welche gleichmäßig über einen zweidimensionalen Raum von 700 mal 350 Längeneinheiten Größe verteilt liegen (Abbildung 4(a)). Jedes Objekt hat eine Größe von 400 Byte.

Das QD-Grid wurde dem herkömmlichen Grid-File gegenübergestellt. Dazu mussten beiden Indexstrukturen zunächst alle Objekte initial indiziern. Das heißt, es wurde ein leerer Index angelegt und alle 10000 Objekte in diesen eingefügt. Anschließend wurden 10000 Punktanfragen an den Index gestellt. Mit Hilfe eines von einem Straßenkartenausschnitt (Abbildung 4(b)) aus Google Maps[2] abgeleiteten Graustufenbildes (Abbildung 4(c)) wurden die Punkte für die Anfragen zufällig aus der Objektmenge in Abbildung 4(a) bestimmt. Das Graustufenbild gab dabei die Anfragewahrscheinlichkeit an der

[2] `http://maps.google.de`



(a) Grid-File (b) QD-Grid

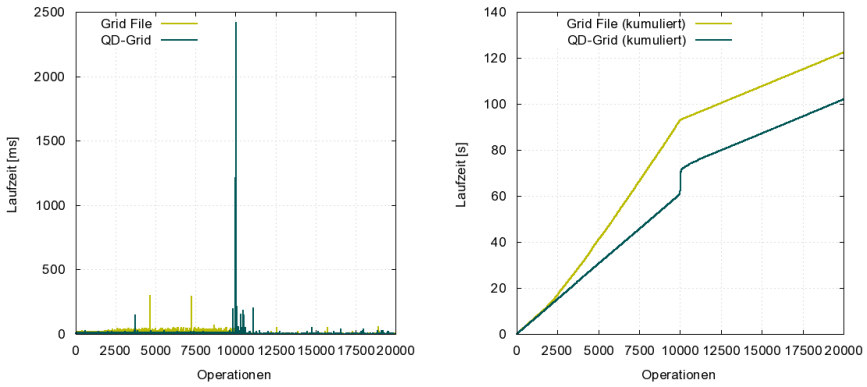
Abbildung 5: Gitterstruktur nach Experimentdurchlauf

jeweiligen Stelle vor. Erzielt wurde eine Anfrageverteilung, welche die Straßen der zu Grunde liegenden Karte reflektiert. Die ungleichmäßig verteilten Anfragepunkte sind in Abbildung 4(d) dargestellt, wobei die Graustufe anzeigt, wie häufig eine Anfrage an diesen Punkt gestellt wird. Je dunkler der Anfragepunkt gezeichnet ist, um so häufiger wurde eine Anfrage an diesem Punkt gestellt.

Implementiert wurden beide Indexstrukturen in Java. Die Implementierung setzte jeweils auf einer einheitlichen Seitenzugriffsschicht auf, welche Zugriff auf eine in 16384 Byte große Seiten unterteilte Datei gibt. Ein Seitenpuffer wurde nicht verwendet. Ausgeführt wurden die Experimente auf einem Arbeitsplatzrechner mit einem Intel Pentium D 3,20 GHz und 2 GB Arbeitsspeicher. Die Softwaresystemumgebung bestand aus Windows Server 2003 SP2 und Sun Java SE Development Kit 6 Update 7.

Abbildung 5 zeigt die Gitterstruktur für beide Implementierungen nach dem Experimentdurchlauf. Es ist gut zuerkennen, wie das herkömmliche Grid-File aufgrund der gleichmäßig verteilten Objekte eine gleichmäßige Gitterstruktur aufbaut (Abbildung 5(a), Objekte im Hintergrund als Punkte dargestellt). Das QD-Grid dagegen orientiert seine Gitterstruktur an den gestellten Anfragen (Abbildung 5(b), Anfragen im Hintergrund als Punkte dargestellt). Deutlich ist zu sehen, dass Bereiche ohne Anfrage vom QD-Grid nicht unterteilt werden. Bei schwarz gezeichneten Zellgrenzen wurden auch die Datenobjekte entsprechend physisch getrennt. Grau gezeichnete Zellgrenzen existieren nur im Gitter, sind jedoch nicht physisch realisiert. Das heißt, benachbarte Zellen, die durch eine graue Linie getrennt sind, verweisen auf den gleichen Datencontainer.

Die Ergebnisse der Laufzeitmessung sind in Abbildung 6 dargestellt, zum einen die Einzellaufzeiten der druchgeführten Einfüge- und Abfrage-Operationen (Abbildung 6(a)) und zum anderen die Entwicklung der kumulierten Laufzeit über den gesamten Experimentverlauf (Abbildung 6(b)). Es ist gut zu sehen, wie das herkömmliche Grid-File deutlich mehr Zeit als das QD-Grid für das initiale Einfügen der 10000 Datenobjekte benötigt, da das QD-Grid beim Einfügen keine Reorganisation der Objekte vornimmt. Ab der ersten Anfrage zeigt das herkömmliche Grid-File jedoch eine konstante Anfragezeit. Für das QD-Grid sind dagegen die ersten Anfragen am aufwändigsten, hier müssen alle Objekte gelesen und mit der Neuaufteilung wieder geschrieben werden. Der Aufwand lohnt sich, da im Folgenden die Anfragebearbeitungszeit sehr schnell sinkt. Sukzessive wird der Teil der Datenmenge, welcher reorganisiert werden muss, kleiner und nach wenigen Reorganisationsoperationen können Anfragen mit der gleichen Geschwindigkeit beantwortet



(a) Einzeln (b) Kumuliert

Abbildung 6: Laufzeiten

werden wie die Anfragen an das herkömmliche Grid-File.

5 Verwandte Arbeiten

Das Konzept der anfragegetriebenen Indizierung steht generell im Bezug zu den Gebieten (1) der Indizierung ortsbezogener Daten (spatial indexing) und (2) der dynamischen Reorganisation von Indexstrukturen. Folglich diskutieren wir zunächst relevante Arbeiten dieser beiden Gebiete und zeigen jeweils anschließend die wesentlichen Unterschiede zum vorgestellten Ansatz auf.

Zur Indizierung ortsbezogener Daten werden in der Regel multidimensionale Zugriffsstrukturen [GG98] mit hierarchischem Aufbau verwendet. Generell unterscheidet man hierbei überlappende und nicht überlappende Strukturen. Ein Beispiel für eine überlappende Struktur stellt der R-Tree [Gut84] dar, wobei sogenannte *minimum bounding rectangles* zur Beschreibung des Datenraums eingesetzt werden. Im Gegensatz dazu beschreiben nicht-überlappende Strukturen wie R+Trees [SRF87], K-D-B-Trees [Rob81], Quad-Trees [SW85] und UB-Trees [RMF$^+$00] Objekte mit einer Menge sich nicht überschneidender Bereiche. Im Rahmen kommerzieller DBMS werden fast ausschließlich derartige hierarchische Strukturen verwendet. So nutzt der MS SQL Server eine hierarchische Grid-Struktur der festen Höhe vier [FFN$^+$08], während in Oracle R-Trees und Quad-Trees zum Einsatz kommen. Neben den Indexstrukturen mit hierarchischem Aufbau kommen auch nicht-hierarchische Strukturen wie das Grid-File [NHS84] zum Einsatz, wobei gerade die dynamische Reorganisation des Zugriffsverzeichnisses ein Problem darstellt. All die vorgestellten Index-Strukturen verbindet, dass diese entweder statisch für eine Datenmenge (das Optimum für den statischen Fall wurde beispielsweise in [PSW95] untersucht) erstellt oder dynamisch mit eintreffenden Updates aktualisiert werden. Der in diesem Papier vorgestellte Ansatz unterscheiden sich von diesen Strukturen dadurch, das

jegliche Reorganisation des Index nur durch entsprechende Anfragen bewirkt wird.

Des Weiteren stellt die dynamische Reorganisation von Indexstrukturen ein für unsere Arbeit relevantes Gebiet dar. Die Kernidee der dynamischen Reorganisation von Indexstrukturen hinsichtlich gestellter Bereichsanfragen ist nicht neu und wurde bereits unter dem Begriff *database cracking* [KM05, IKM07] näher betrachtet. Weiterhin folgt auch das Konzept der sogenannten *partitioned B-Trees* [Gra03] einer ähnlichen Vorgehensweise der dynamischen Reorganisation von Partitionen. Diese Arbeiten adressieren bislang aber ausschließlich eindimensionale Indexstrukturen und wurden somit bislang noch nicht für die Indizierung räumlicher Daten eingesetzt. Neben der dynamischen Reorganisation von Indizes existieren auch Gemeinsamkeiten mit adaptiven Datenstrukturen für *moving objects* [TP02, CML04] und *spatial-temporal queries* [CMC04] (APR-tree) als auch für zustandsbehaftetes *information filtering* [DFK05]. Diese adaptive Indexverwaltung wird wiederum jedoch nur durch Updates (neue Objekte oder Bewegung der Objekte), nicht jedoch – wir in unserem Fall – durch Anfragen initiiert.

Zusammenfassend ist also einzuschätzen, dass sowohl die Indizierung ortsbezogener Daten mit Hilfe von Grid-Hierarchien als auch die dynamische Reorganisation von Indexstrukturen wohlbekannte Methoden sind. Im Rahmen dieser Arbeit werden jene Aspekte jedoch erstmals derart kombiniert, dass lesende Anfragen implizit den Aufbau und die Reorganisation des ortsbezogenen (und damit mehrdimensionalen) Index bewirken.

6 Zusammenfassung

Dieses Papier stellt das QD-Grid vor, eine räumliche Indexstruktur deren Aufbau von Anfragen getrieben wird. Daten, die zur Beantwortung von Anfragen gelesen werden müssen, werden im QD-Grid während der Anfrageverarbeitung reorganisiert, so dass zukünftige, ähnliche Anfragen von dem bereits geleisteten Leseaufwand profitieren. Somit erlaubt es das QD-Grid, die Indizierung an der Arbeitslast zu orientieren. Bereiche mit häufig gestellten Anfragen werden feingranularer indiziert als Bereiche von geringem bzw. keinem Interesse.

Der vorgestellte Ansatz soll noch weiterentwickelt werden. So scheint es sinnvoll, die Reorganisation der Indexstruktur nicht bei jeder Anfrage zu triggern, sondern beispielsweise durch die Definition von Grenzwerten zu verzögern. Weiterhin ist es bei einer selbstorganisierenden Indexstruktur erforderlich, sich nicht nur einer Arbeitslast initial anzupassen, sondern auch langfristige Änderungen der Arbeitslast nachzuvollziehen. Für eine solche Anpassung müssen beim QD-Grid beispielsweise benachbarte Zellen, auf welche nur noch selten Anfragen gestellt werden, wieder zusammengefügt werden.

Literatur

[CMC04] Hyung-Ju Cho, Jun-Ki Min und Chin-Wan Chung. An adaptive indexing technique using spatio-temporal query workloads. *Information & Software Technology*, 46(4):229–241, 2004.

[CML04] Wonik Choi, Bongki Moon und Sukho Lee. Adaptive cell-based index for moving objects. *Data Knowl. Eng.*, 48(1):75–101, 2004.

[DFK05] Jens-Peter Dittrich, Peter M. Fischer und Donald Kossmann. AGILE: Adaptive Indexing for Context-Aware Information Filters. In *SIGMOD Conference*, Seiten 215–226, 2005.

[FFN+08] Yi Fang, Marc Friedman, Giri Nair, Michael Rys und Ana-Elisa Schmid. Spatial indexing in microsoft SQL server 2008. In *SIGMOD Conference*, Seiten 1207–1216, 2008.

[GG98] Volker Gaede und Oliver Günther. Multidimensional Access Methods. *ACM Comput. Surv.*, 30(2):170–231, 1998.

[Gra03] Goetz Graefe. Sorting And Indexing With Partitioned B-Trees. In *CIDR*, 2003.

[Gut84] Antonin Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In *SIGMOD Conference*, Seiten 47–57, 1984.

[IKM07] Stratos Idreos, Martin L. Kersten und Stefan Manegold. Database Cracking. In *CIDR*, Seiten 68–78, 2007.

[KM05] Martin L. Kersten und Stefan Manegold. Cracking the Database Store. In *CIDR*, Seiten 213–224, 2005.

[NHS84] Jürg Nievergelt, Hans Hinterberger und Kenneth C. Sevcik. The Grid File: An Adaptable, Symmetric Multikey File Structure. *ACM Trans. Database Syst.*, 9(1):38–71, 1984.

[PSW95] Bernd-Uwe Pagel, Hans-Werner Six und Mario Winter. Window Query-Optimal Clustering of Spatial Objects. In *PODS*, Seiten 86–94, 1995.

[RMF+00] Frank Ramsak, Volker Markl, Robert Fenk, Martin Zirkel, Klaus Elhardt und Rudolf Bayer. Integrating the UB-Tree into a Database System Kernel. In *VLDB*, Seiten 263–272, 2000.

[Rob81] John T. Robinson. The K-D-B-Tree: A Search Structure For Large Multidimensional Dynamic Indexes. In *SIGMOD Conference*, Seiten 10–18, 1981.

[SRF87] Timos K. Sellis, Nick Roussopoulos und Christos Faloutsos. The R+-Tree: A Dynamic Index for Multi-Dimensional Objects. In *VLDB*, Seiten 507–518, 1987.

[SW85] Hanan Samet und Robert E. Webber. Storing a Collection of Polygons Using Quadtrees. *ACM Trans. Graph.*, 4(3):182–222, 1985.

[TP02] Yufei Tao und Dimitris Papadias. Adaptive Index Structures. In *VLDB*, Seiten 418–429, 2002.