Verarbeitung von kontinuierlichen Anfragen auf räumlichen Datenbanken und Indexierung von mobilen Objekten [1]

Matthias Burkhart

Betreuer:
Dipl.-Inf. Jürgen Göres
Lehrgebiet Informationssysteme
TU Kaiserslautern

[1] Arbeit im Rahmen des Seminars „Mobile and Context-aware Database Technologies and Applications“

1 Einleitung

Mobile Geräte wie Mobiltelefone, Notebooks oder PDAs erfreuen sich immer größerer Beliebtheit. Bereits jetzt sind sie für viele Menschen wichtiger Bestandteil des alltäglichen Lebens und werden ständig mitgeführt. Ihre große Popularität lässt sich darauf zurückführen, dass sie es in Verbindung mit drahtlosen Netzwerken ermöglichen, mit der Außenwelt zu kommunizieren. Dank des technologischen Fortschritts steigen Rechenleistung und Bandbreite der tragbaren Geräte, wohingegen Größe und Gewicht sinken. Immer komplexere und ressourcenintensivere Dienste lassen mobile Geräte zum Alleskönner mutieren.

Beschränkte sich der frühere Funktionsumfang von Mobiltelefonen auf das Telefonieren und den Versand von SMS, so geht man heute damit ins Internet, um aktuelle Informationen wie etwa Börsenkurse oder Sportergebnisse abzurufen, verwaltet Kontaktdaten und Termine, hört Musik oder zeichnet Videos auf.

Für viele dieser Anwendungen spielt der *Kontext* eine wichtige Rolle. Dey und Abowd [7] definieren Kontext als „alle Informationen, die für die Interaktion zwischen einem Benutzer und einer Applikation als relevant erachtet werden“. Damit sind Informationen über den Ort oder Objekte und Personen in der Umgebung gemeint, aber auch über die Situation des Benutzers und die Applikation selbst.

Kontextinformationen bieten den Softwareentwicklern völlig neue Möglichkeiten. So könnte ein Musikprogramm anhand des Musikgeschmacks des Benutzers eine Playlist generieren und diese automatisch anpassen, falls sich die Gemütslage des Benutzers ändert. Ein anderes Beispiel wäre ein Handy, das über ein Positionierungssystem weiß, dass sich der Benutzer gerade im Schlafzimmer befindet. Aufgrund der Uhrzeit und der Raumhelligkeit folgert es, dass der Benutzer schläft. Bei eingehenden Anrufen erkennt das Telefon selbständig, ob der Benutzer lieber weiterschlafen will (in diesem Fall stellt sich das Handy auf lautlos) oder ob ihm der Anruf so wichtig ist, dass er dafür geweckt werden will (das Telefon klingelt).

Ziel der Kontextbenutzung ist es, dem Benutzer neue Serviceleistungen zu bieten und die Benutzerfreundlichkeit von mobilen Geräten zu erhöhen. Ein neuer Service hat wenig Sinn, wenn der Aufwand zur Konfiguration höher ist, als der dadurch gewonnene Nutzen. Neue Funktionen sollten also möglichst automatisch im Sinne des jeweiligen Benutzers ausgeführt werden.

Bisher sind Ortsinformationen die mit Abstand am häufigsten benutze Form von Kontext. Bekanntes Beispiel sind Navigationssysteme. Diese weisen den Weg zu einem gewünschten Ziel, wie etwa einer Stadt oder einer Straße. Städte ändern nicht ihre Position und neue Straßen werden selten gebaut. Das führt dazu, dass selten Änderungsoperationen auf den von der Anwendung benötigten Daten durchgeführt werden müssen.

Problematisch wird es, wenn die Ortsinformationen von *mobilen Objekten* in einer Datenbank verwaltet werden sollen, wie es für viele neuartige Anwendungen erforderlich ist. Mobile Objekte verursachen hohe Updateraten, womit traditionelle räumliche Datenbankindizes nicht zurechtkommen.

Mobile Objekte stellen auch für die Bearbeitung von räumlichen Anfragen erhebliche Schwierigkeiten dar, da Ergebnisse, die eben noch gültig waren im nächsten Moment schon falsch sein können.

In dieser Ausarbeitung werden Lösungsansätze für diese Probleme vorgestellt. Kapitel 2 beschäftigt sich mit der Verarbeitung von kontinuierlichen Anfragen auf räumlichen Datenbanken. In Kapitel 3 werden Verfahren zur effizienten Indexierung von beweglichen Objekten vorgestellt, bevor Kapitel 4 mit einer kurzen Zusammenfassung abschließt.

2 Verarbeitung von kontinuierlichen Anfragen auf räumlichen Datenbanken

Anfragen auf Datenbanken waren bisher eher statischer Natur. So können Datenbankmanagementsysteme Anfragen beantworten wie z. B.: „Den wievielten Platz belegte der 1. FC Kaiserslautern in der Saison 74/75?“. Räumliche Anfragen, wie etwa „Welcher Fluß fließt durch Passau?“ oder „In welchem Bundesland liegt Köln?“, können mit SQL nur umständlich gestellt werden, lassen sich aber mit einer GIS-Erweiterung realisieren.

Durch das gesellschaftliche Phänomen Mobiltelefon trägt heutzutage nahezu jeder, nahezu ständig einen mobilen Rechner bei sich, über den er mit seiner Umgebung kommunizieren kann. Begünstigt durch Fortschritte in der Positionsbestimmung, höhere Übertragungsraten bei Funknetzen, sowie ein immer besser werdendes Preis-Leistungsverhältnis bei mobilen Geräten, entsteht eine Infrastruktur für neue *ortsabhängige* Services.

Eine heute schon weitverbreitete, ortsabhängige Anwendung ist das Navigationssystem. Es bedient sich der GPS-Technologie und verschafft Autofahrern einen enormen praktischen Nutzen, der bald wohl so selbstverständlich sein dürfte wie die Servolenkung heutzutage schon ist.

Zukünftig werden noch viele weitere ortsabhängige Leistungen hinzukommen. Zur Bereitstellung ihrer Leistung benötigen sie verschiedene Arten von Anfragen auf räumliche Daten. Die wichtigsten Anfragearten werden an dieser Stelle kurz vorgestellt.

2.1 Anfragearten auf räumliche Daten

Die beiden am häufigsten benutzten Anfragearten auf räumliche Datenbanken sind die *Nearest-Neighbour-Anfrage (Nearest-Neighbour Query)* und die *Bereichsanfrage (Range Query)*.

Damit könnte sich ein PKW-Fahrer während der Autofahrt die fünf nächstgelegenen Restaurants anzeigen lassen (Nearest-Neighbour-Anfrage) oder alle Hotels im Umkreis von fünf Kilometern (Bereichsanfrage). Betrachten wir dazu Abbildung 1.

In a) liefert eine Nearest-Neighbour-Anfrage q Objekt O3 als Ergebnis zurück. Die Bereichsanfrage in b) umfasst alle Objekte, die sich im angefragten Bereich q befinden. Hier also O5-O8. Diese Anfragen unterscheiden sich von traditionellen

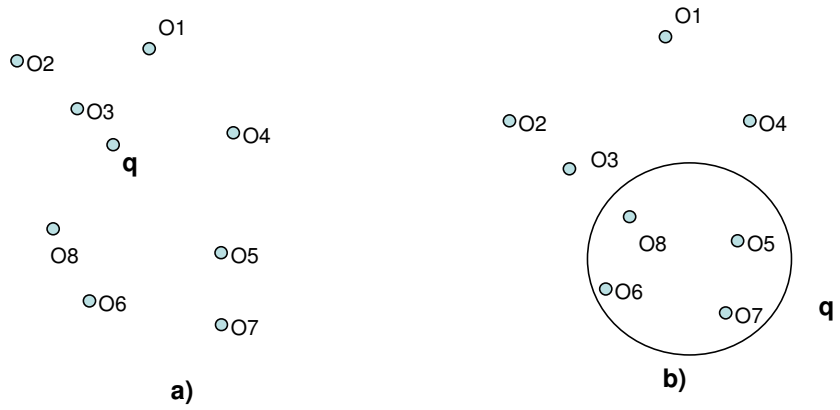

Abbildung 1. Nearest-Neighbour-Anfrage und Bereichsanfrage

Anfragen insofern, dass es für die Anfrage relevant ist, wo sie gestellt wurde. Angenommen zwei Autofahrer in München bzw. Hamburg suchen die nächstgelegene Tankstelle, dann werden sie auf Grund ihrer unterschiedlichen Position selbstverständlich auch unterschiedliche Ergebnisse zurückgeliefert bekommen.

Anfragen auf spatio-temporale Datenbanken bieten außerdem die Möglichkeit, das Suchergebnis nicht nur räumlich, sondern auch zeitlich einzuschränken. Mögliches Anwendungsfeld könnte hierbei der Personentransport sein. Eine nützliche Information wäre, wann sich ein Bus mit passender Route in Benutzernähe befindet.

Betrachten wir nun den Fall, dass bei einem PKW der Kraftstoff zur Neige geht und der Fahrer explizit einen Dienst aufruft, der ihm die nächstgelegene Tankstelle anzeigt. Hier wird typischerweise eine einzelne Anfrage gestellt, welche mit Hilfe einer räumlichen Zugriffsmethode (s. Kapitel 3) beantwortet wird und anschließend terminiert. Erweitert man den Service dahingegen, dass während der kompletten Autofahrt immer die nächstgelegene Tankstelle angezeigt wird, so spricht man von einer *kontinuierlichen Anfrage*.

Kontinuierliche Anfragen, die von bewegten Objekten ausgehen, haben aufgrund ihrer Mobilität die Eigenschaft, dass ein Ergebnis, das gerade noch richtig war, im nächsten Moment schon wieder falsch sein kann. Eine konservative Vorgehensweise, die immer ein richtiges Ergebnis liefert, schickt bei jedem Positionswechsel eine neue Anfrage an den Server.

2.2 Reduzierung der Anfragehäufigkeit

Um diesem Anfrage- und Netzwerk-Overhead entgegenzuwirken, haben Zhang et al. [8] ein Verfahren für *statische Objekte* entwickelt, welches zusätzlich zum Anfrageergebnis noch einen Bereich bestimmt, in dem sich das Ergebnis nicht verändert (*Gültigkeitsbereich*). Erst beim Verlassen dieses Gültigkeitsbereichs muss eine neue Anfrage gestellt werden. Dazu werden sogenannte *Einflussobjekte*

ermittelt, mit denen der Client entscheiden kann, ob er sich noch im Gültigkeitsbereich befindet. Basierend auf einem beliebigen räumlichen Datenbankindex (etwa dem R-Baum [5]) lässt sich das Verfahren sowohl für Nearest-Neighbour-Anfragen als auch für Bereichsanfragen anwenden. Betrachten wir zuerst die Vorgehensweise bei einer Nearest-Neighbour-Anfrage.

Ermittlung der Einflussobjekte bei Nearest-Neighbour-Anfragen. Das Nearest-Neighbour-Objekt *nn* kann mit einem traditionellen Nearest-Neighbour-Algorithmus leicht gefunden werden. Der Gültigkeitsbereich für die Nearest-Neighbour-Anfrage q ist dann ein Halbraum, in dem jeder Punkt näher bei *nn* liegt, als bei irgendeinem anderen Objekt im Datenuniversum, wobei das Datenuniversum das komplette räumliche Gebiet ist, das für die Anfrage als relevant erachtet wird.

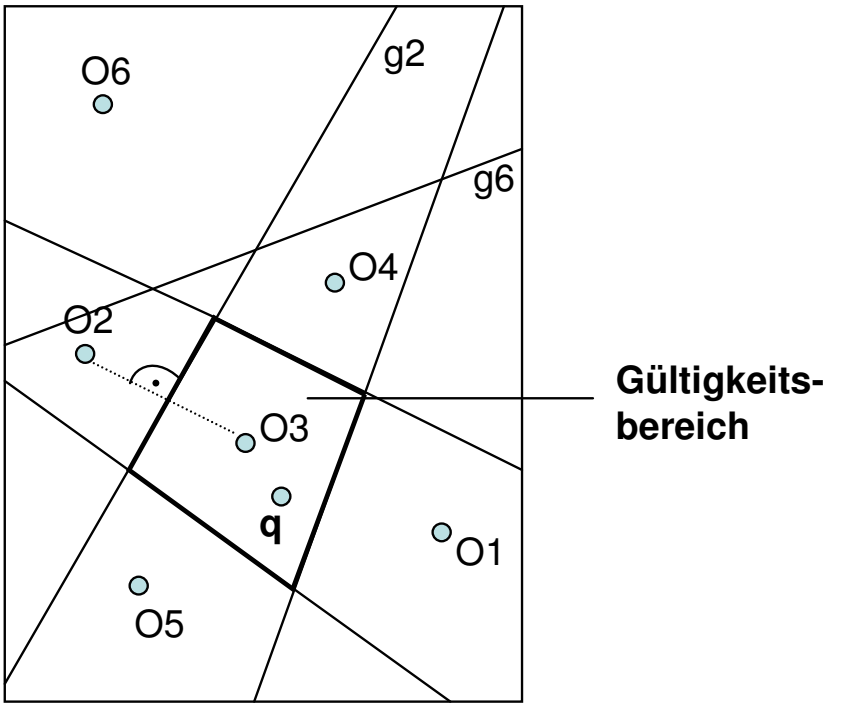

Abbildung 2. Nearest-Neighbour-Anfrage im zweidimensionalen Raum

Betrachten wir dazu Abbildung 2. Der Client stellt an Position q eine Nearest-Neighbour-Anfrage. Der Datenbankindex findet O3 als Nearest-Neighbour. Wie groß der Gültgkeitsbereich für Anfrage q ist, hängt von den umliegenden Objekten ab. Zwischen O2 und O3 liegt die Gerade g2. Sie ist die Mittelsenkrechte

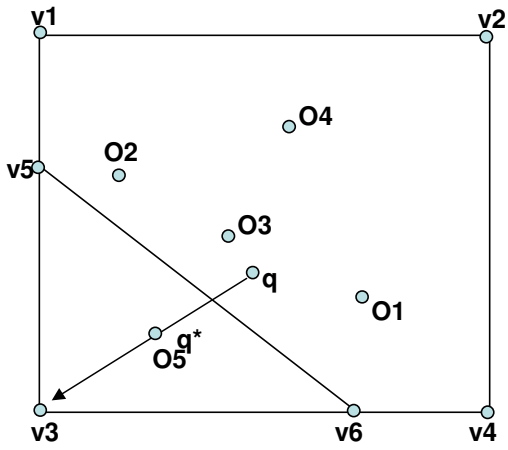

Abbildung 3. Verfahren zur Bestimmung der Einflussobjekte

zwischen diesen beiden Punkten und teilt das Datenuniversum in zwei Halbebenen. Alle Punkte, die in der gleichen Halbebene wie O3 liegen, haben einen kleineren Abstand zu O3 als zu O2 und umgekehrt. So kann man für jedes Objekt im Datenuniversum vorgehen und erhält jedesmal eine Halbebene, die O3 enthält. Der Durchschnitt aller so gebildeten Halbebenen bildet den Gültigkeitsbereich für Anfrage q.

Objekte, deren Mittelsenkrechte ein Stück vom Rand des Gültigkeitsbereichs bilden, werden Einflussobjekte genannt. In unserem Beispiel wären das O1, O2, O4 und O5. O6 steuert keine Kante zum Rand der Gültigkeitsregion bei. Es hat deswegen keinen Einfluss auf die Ausmaße des Gültigkeitsbereichs und ist somit auch kein Einflussobjekt. Die Menge der Einflussobjekte bezeichnen wir $S_{inf}(q)$. Sie werden mit folgendem iterativen Algorithmus berechnet:

Zu Beginn der Berechnung wird die Gültigkeitsregion als das komplette Datenuniversum angenommen, das durch die vier Ecken v1 bis v4 bestimmt ist (s. Abbildung 3). Diese wird durch neu hinzukommende Einflussobjekte schrittweise verkleinert und entspricht am Ende dem tatsächlichen Gültigkeitsbereich.

Um Einflussobjekte zu finden, wird eine Ecke der aktuellen Gültigkeitsregion willkürlich ausgewählt (hier v3). Von Position q aus wird eine *zeitparametrisierte Anfrage* q* in Richtung v3 gestellt.

Zeitparametrisierte Anfragen [9] haben eine Richtung, in die sie sich mit konstanter Geschwindigkeit bewegen. Ihre Ergebnisse haben die Form $\langle R, T, C \rangle$, wobei R das konventionelle Ergebnis (der Nearest-Neighbour bei einer zeitparametrisierten Nearest-Neighbour-Anfrage) zum Anfragezeitpunkt darstellt. T ist die Zeitdauer, bis sich das Ergebnis verändert und C ist die Objektmenge, die zu einer Änderung der Ergebnismenge führt.

Für das vorgestellte Verfahren, wird die Zeitkomponente T nicht benötigt. Die zeitparametrisierte Nearest-Neighbour-Anfrage q* liefert für die Menge $C = \{-O3, +O5\}$. Dies bedeutet, dass nach einer gewissen Zeit O3 von O5 als Nearest-

Neighbour abgelöst wird. O3 wird von der bisherigen Ergebnismenge $R = \{O3\}$ abgezogen und O5 hinzugefügt.

Allgemein gilt, dass sobald sich ein Objekt näher an der zeitparametrisierten Anfrage befindet als der Nearest-Neighbour (vorausgesetzt so ein Objekt existiert), ein Einflussobjekt gefunden wurde. In unserem Beispiel ist O5 das gefundene Einflussobjekt. Dieses wird zu $S_{inf}(q)$ hinzugefügt. Die zu O3 und O5 gehörende Mittelsenkrechte g5 bildet wieder zwei Halbebenen, von denen diejenige, die nicht O3 enthält, vom aktuellen Gültigkeitsbereich abgezogen wird. Mit v5 und v6 entstehen dadurch zwei neue Ecken und der neue Gültigkeitsbereich setzt sich dann aus den Ecken v1, v2, v4, v5 und v6 zusammen. Aus diesen wird wiederum beliebig eine ausgewählt und erneut eine zeitparametrisierte Anfrage in deren Richtung gestellt. Wird dabei kein Einflussobjekt gefunden, so wird die Ecke markiert. Markierte Ecken werden bei der nächsten Auswahl nicht mehr berücksichtigt und der Algorithmus terminiert, wenn alle Ecken markiert sind. Ein Beweis, dass dieses Verfahren alle Einflussobjekte korrekt auswählt, findet man in [8].

Auf Client-Seite kann mit dem Nearest-Neighbour und der Menge $S_{inf}(q)$ die Gültigkeitsregion berechnet werden. Dazu wird wie oben beschrieben der Durchschnitt aller Halbebenen gebildet.

Für Bereichsanfragen existiert ein ähnliches Verfahren.

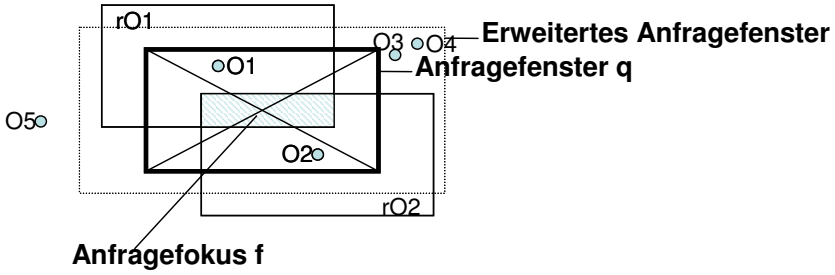

Abbildung 4. Innerer Gültigkeitsbereich

Ermittlung der Einflussobjekte bei Bereichsanfragen. Abbildung 4 zeigt nun eine Bereichsanfrage q. Die Objekte im Anfragefenster sind das Ergebnis der Bereichsanfrage. Sie können mit einem traditionellen Datenbankindex gefunden werden und werden innere Punkte genannt (Objekte O1 und O2 im Beispiel). Die Objekte außerhalb des Anfragefensters nennt man äußere Punkte (Objekte O3, O4 und O5). Der Anfragefokus ist der Mittelpunkt des Anfragefensters.

Auch hier ist der Gültigkeitsbereich die Fläche, in der sich das Anfrageergebnis nicht ändert. In Abbildung 4 ist für jeden inneren Punkt die *Minkowski-Region* eingezeichnet. Diese hat die gleiche Größe wie das Anfragefenster mit dem jeweiligen inneren Punkt im Fokus (s. rO1 und rO2). Bewegt sich der Fo-

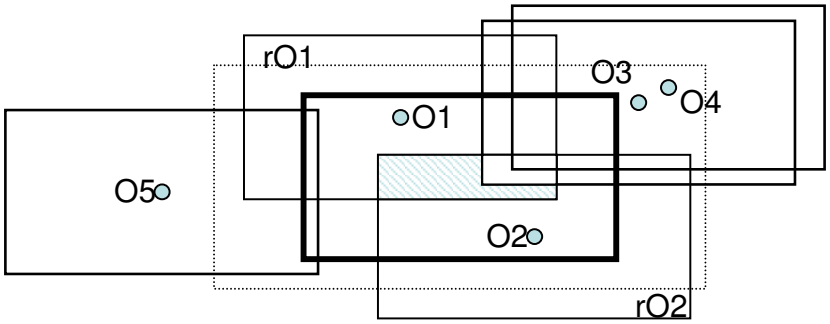

Abbildung 5. Äußere Einflussobjekte

kus der Bereichsanfrage in der Minkowski-Region eines Objektes, so ist das Objekt immer Teil des Ergebnisses der Anfrage. Bewegt sich der Anfragefokus im Schnitt der Minkowski-Regionen aller inneren Objekte, so befinden sich alle inneren Objekte im Anfragefenster. Dieser Schnitt wird *innerer Gültigkeitsbereich* genannt.

Dies ist noch nicht unser gesuchter Gültigkeitsbereich, da berücksichtigt werden muss, dass neue Objekte hinzukommen können, falls sich der Anfragefokus bewegt. Um diese neuen Objekte zu finden, wird das Anfragefenster erweitert, indem oben und unten jeweils die halbe Höhe und rechts und links jeweils die halbe Breite der inneren Gültigkeitsregion hinzugefügt wird (s. Abbildung 4). Mit dem erweiterten Anfragefenster wird eine neue Anfrage gestellt und von der so erhaltenen Menge von Objekten wird die Menge der inneren Objekte abgezogen. In unserem Beispiel erhält man hierbei O3 und O4. Die Minkowski-Regionen dieser äußeren Punkte schneiden nun die innere Gültigkeitsregion (s. Abbildung 5). Zieht man die Schnittflächen von der inneren Gültigkeitsregion ab, so erhält man die endgültige Gültigkeitsregion, in der sich die Anfrage bewegen kann, ohne dass sich das Ergebnis ändert. Erst beim Verlassen dieser endgültigen Gültigkeitsregion muss eine neue Anfrage gestellt werden.

Auch bei Bereichsanfragen wird die Größe der endgültigen Gültigkeitsregion von Einflussobjekten bestimmt. Es wird zwischen inneren und äußeren Einflussobjekten unterschieden. Potentielle innere Einflussobjekte sind diejenigen, die mindestens eine Kante zum Rand des inneren Gültigkeitsbereichs beisteuern. Potentielle äußere Einflussobjekte sind diejenigen, die den inneren Gültigkeitsbereich schneiden. Ein Schnitt von der Minkowski-Region eines potentiellen äußeren Einflussobjekts und des inneren Gültigkeitsbereichs kennzeichnet die Region, in dem sowohl alle inneren Objekte, als auch das potentielle äußere Einflussobjekt zum Anfrageergebnis gehören. Um die tatsächlichen Einflussobjekte zu finden, stellt man jetzt wie bei den Nearest-Neighbour-Anfragen zeitparametrisierte Bereichsanfragen in Richtung einer Ecke des inneren Gültigkeitsbereichs. Dabei erhält man entweder ein Einflussobjekt, oder die Ecke wird gekennzeichnet und nicht mehr betrachtet. Als äußeres Einflussobjekt wird das Objekt zurück-

geliefert, dessen Minkowski-Region zuerst von der zeitparametrisierten Anfrage geschnitten wird. Diese Minkowski-Region schneidet den inneren Gültigkeitsbereich und die Schnittmenge wird vom aktuellen Gültigkeitsbereich abgezogen. Dadurch erhält man neue Ecken. Dieser Vorgang wird so lange wiederholt, bis alle Ecken markiert sind. Alle Einflussobjekte werden so gefunden (Beweis dazu in [8]).

In Abbildung 5 sehen wir die potentiellen äußeren Einflussobjekte O3 und O4. Im Gegensatz zu O5 schneiden ihre Minkowski-Regionen den inneren Gültigkeitsbereich. O5 ist deshalb kein potentielles äußeres Einflussobjekt. Bei einer zeitparametrisierten Anfrage in Richtung der rechten oberen Ecke des inneren Gültigkeitsbereichs wird O3 als tatsächliches äußeres Einflussobjekt zurückgeliefert. Der Schnitt der Minkowski-Region von O4 mit dem inneren Gültigkeitsbereich wird komplett vom Schnitt der Minkowski-Region von O3 mit dem inneren Gültigkeitsbereich überdeckt. Bevor das Anfrageergebnis von O4 verändert wird, muss es also bereits von O3 verändert worden sein. O3 ist somit ein tatsächliches äußeres Einflussobjekt, O4 hingegen nur ein potentielles.

Der Client bekommt als Ergebnis die inneren Objekte zusammen mit den Einflussobjekten geliefert. Um zu überprüfen, ob die aktuelle Anfrage noch gültig ist, wird für die inneren Einflussobjekte geprüft, ob sich der Anfragefokus f noch in ihren Minkowski-Regionen befindet. Desweiteren darf sich f nicht in einer Minkowski-Region eines äußeren Einflussobjekts befinden, da sonst das äußere Einflussobjekt zum Anfrageergebnis hinzukommen würde. Solange dies gilt, ist keine neue Anfrage notwendig.

Nachteil dieses Verfahrens ist seine Beschränkung auf statische Datenobjekte.

2.3 Reduzierung der Anfragekosten

Mokbel und Aref bieten mit dem *Generic Progressive Algorithm (GPAC)* [10] ein Framework für die Beantwortung kontinuierlicher Anfragen auf mobile Objekte. Der Ansatz ist in dem Sinne generisch, dass sich mit dem Algorithmus eine Vielzahl von spatio-temporalen Anfragen realisieren lassen. Die Bezeichnung als Framework kommt daher, dass bestimmte Methoden des Algorithmus auf den jeweiligen Anfragetyp angepasst werden müssen. GPAC verarbeitet sowohl statische als auch kontinuierliche Anfragen (etwa kontinuierliche Bereichsanfragen- und Nearest-Neighbour-Anfragen). Es ist anzumerken , dass das Verfahren darauf basiert, dass die mobilen Objekte regelmäßig Updates bezüglich ihrer Position versenden.

Die meisten existierenden Verfahren zur Bearbeitung von spatio-temporalen Anfragen basieren darauf, dass die von den mobilen Objekten gesendeten Ortsinformationen indexiert und auf einem externen Speichermedium abgelegt werden. Jedoch führen die hohen Zugriffszeiten des Externspeichers unter anderem dazu, dass diese Verfahren in der Praxis mit den hohen Updateraten der mobilen Objekte nicht zurechtkommen. Um schnell und effizient arbeiten zu können, materialisiert GPAC die Daten nicht auf dem Externspeicher. Stattdessen werden eingehende Datenströme direkt im Hauptspeicher verarbeitet.

GPAC verknüpft jede kontinuierliche Anfrage mit einem mobilen Objekt, welches den Fokus der Anfrage bildet.

Ein spatio-temporaler Anfragetyp kann als Prädikat aufgefasst werden. GPAC speichert die Objekte, die das Anfrageprädikat erfüllen in einer Ergebnismenge ab. Sendet nun ein mobiles Objekt O ein Tupel mit neuen Objektinformationen, so überprüft GPAC , ob O bereits in der bisherigen Ergebnismenge E_{alt} vorhanden ist und ob das Datentupel dem Anfrageprädikat P genügt. Dabei sind vier Fälle zu unterscheiden:

1. $O \in E_{alt}$ und O erfüllt P: Das Objekt war bereits im Anfrageergebnis und erfüllt immer noch das Anfrageprädikat. Es ist nichts zu tun.
2. $O \in E_{alt}$ und O erfüllt P nicht: Das Objekt ist im Anfrageergebnis, dürfte aber nicht mehr darin sein, da es das Prädikat nicht mehr erfüllt. Dem Benutzer muss ein negatives Update -O geschickt werden. Das neue Ergebnis E_{neu} berechnet sich aus $E_{neu} = E_{alt} - O$.
3. $O \notin E_{alt}$ und O erfüllt P: Das Objekt ist nicht im Anfrageergebnis, sollte aber darin sein, da es das Anfrageprädikat erfüllt. Also muss ein positives Update +O geschickt werden. $E_{neu} = E_{alt} + O$.
4. $O \notin E_{alt}$ und O erfüllt P nicht: Wie im ersten Fall ist hier nichts zu tun, da O nicht im Anfrageergebnis ist und auch nach seinem Update das Prädikat nicht erfüllt.

GPAC versucht, so wenig Objekte wie möglich im Hauptspeicher zu halten. Ein naiver Ansatz würde alle Objekte, welche das Anfrageprädikat nicht erfüllen aus dem Hauptspeicher entfernen.

Das Problem, das dieses Vorgehen mit sich bringen würde, wird in Abbildung 6 an einem Beispiel deutlich.

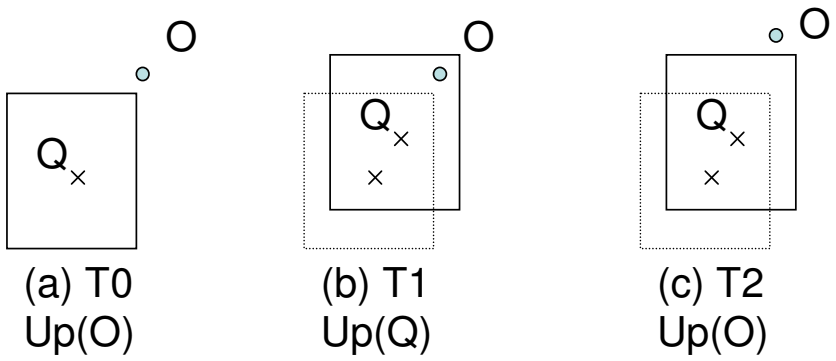

Abbildung 6. Kritische Situation bei mobilen Anfragen.

In (a) sehen wir eine mobile Bereichsanfrage Q. Zum Zeitpunkt T0 sendet Objekt O seine aktuelle Position. Da diese sich aber nicht im Anfragefenster

von Q befindet (das Prädikat nicht erfüllt), würde sie nach obiger Annahme verworfen werden. Zum Zeitpunkt T1 (b) sendet das der Anfrage Q zugeordnete Objekt ein Update. Seine Position hat sich in die Richtung des Objekts O verschoben. Objekt O liegt nun im Anfragefenster und müsste eigentlich zum Anfrageergebnis hinzugefügt werden. Die Positionsinformationen von P wurden jedoch nicht gespeichert und daher besteht auch keine Chance für den Algorithmus, Objekt Q über das neue korrekte Ergebnis zu informieren. Später (T2) hat sich O wieder außerhalb des Anfragefenster bewegt und erfüllt nicht mehr länger das Anfrageprädikat. O wird erneut verworfen.

Wie wir sehen, kann es so zu falschen Ergebnissen kommen. O lag zwischen T1 und T2 im Anfragefenster, tauchte aber zu keinem Zeitpunkt im Ergebnis auf.

Es existiert ein kritischer Bereich, in dem sich Objekte befinden können, die zwar das Anfrageprädikat erfüllen, jedoch im Hauptspeicher nicht mehr existent sind. max ist die maximale Strecke, die ein Objekt zwischen zwei Positionsupdates zurücklegen kann. Um Unkorrektheiten auszuschließen, werden daher alle Objekte, die nicht weiter als $2 \cdot max$ vom Anfragefenster entfernt sind, in einem *Cache* gelagert.

Unsere obigen vier Fälle müssen mit dem neuen Wissen ergänzt werden. Es gilt, zusätzlich folgende Fälle zu beachten:

- Q bewegt sich: Dabei aktualisiert sich nicht nur das Anfragefenster, sondern auch der kritische Bereich, der vom Cache abgedeckt werden muss. Für alle Objekte im Cache wird getestet, ob sie sich durch die Bewegung mittlerweile im Anfragefenster befinden. Dann werden sie zur Ergebnismenge hinzugefügt und ein positives Update zum Benutzer geschickt. Befinden sie sich jedoch nicht mehr in der Cache-Region, so werden sie aus dem Cache gelöscht. Für die Objekte aus der Ergebnismenge wird geprüft, ob sie weiter das Prädikat erfüllen. In diesem Fall geschieht nichts. Andernfalls werden sie entweder in den Cache verschoben, oder komplett aus dem Hauptspeicher gelöscht, wenn sie sich nicht mehr in der Cache-Region befinden. Ein negatives Update wird zum Benutzer geschickt.
- $O \notin E_{alt}$ und O erfüllt P: Falls $O \in$ Cache, muss O aus dem Cache gelöscht werden.
- $O \in E_{alt}$ und O erfüllt P nicht: Es muss überprüft werden, ob sich O noch in der Cache-Region befindet. Falls ja, muss es in den Cache gespeichert werden.
- $O \notin E_{alt}$ und O erfüllt P nicht: Ist O in der Cache-Region, so wird überprüft, ob O bereits im Cache gespeichert ist. Falls nicht, wird es hinzugefügt. Ist O nicht in der Cache-Region, so wird es aus dem Cache entfernt, falls es darin gespeichert war.

GPAC bearbeitet die Anfragen auf eine iterative Weise, da immer nur die Änderungen vom zuletzt gültigen Ergebnis zum Benutzer gesendet werden (positive und negative Updates). Dadurch wird die Übertragungsdauer reduziert und das Übertragungsmedium entlastet.

3 Indexierung von mobilen Objekten

Im vorigen Kapitel haben wir verschiedene Anfragearten und Möglichkeiten zur Reduktion der Anfragehzahl und Anfragekosten auf Datenbanken für mobile Objekte kennen gelernt. Damit diese Anfragen in möglichst kurzer Zeit beantwortet werden können, müssen effiziente Indizes existieren, die für große und dynamische Datenmengen ausgelegt sind. Traditionelle Datenbankindizes für räumliche Informationen wie der R-Baum [5] oder das GridFile [4] sind für die hohen Updateraten mobiler Objekte nicht geeignet, da sie von eher statischen Daten mit wenigen Veränderungen ausgehen. Es gibt zahlreiche Lösungsansätze, die traditionellen Indizes den neuen Anforderungen anzupassen, von denen im Folgenden zwei vorgestellt werden.

3.1 Reduzierung der Updatehäufigkeit

Die *Broad B^x indexing technique (BB^x-Index)* von Lin et al. [1] indexiert vergangene, aktuelle und zukünftige Positionen von mobilen Objekten. Dies ermöglicht Suchanfragen, die sich nicht nur räumlich, sondern auch zeitlich einschränken lassen.

Der BB^x-Index unterteilt die Zeit in Intervalle gleicher Länge Δt_{mu}. Dies ist die Zeit, die maximal vergehen darf, bevor ein mobiles Objekt ein Update seiner Position an den Server schicken muss.

Positionen von Objekten lassen sich auf verschiedene Arten darstellen. Einige Verfahren machen mit Hilfe von Funktionen, abhängig von der aktuell indexierten Position, Aussagen über voraussichtliche zukünftige Positionen. Ein Update erfolgt, sobald die Abweichung der durch die Funktion errechnete Position und der tatsächlichen Objektposition einen bestimmten Wert x_{dif} übersteigt. x_{dif} bestimmt also die Präzision, mit der ein Index arbeitet. Sie hängt stark von der jeweiligen Anwendung ab. Wettervorhersagen benötigen zum Beispiel geringere Genauigkeit als Navigationssysteme. Ein geringeres toleriertes x_{dif} führt zu einer größeren Genauigkeit, aber auch zu einer Erhöhung der Updatehäufigkeit.

Die einfachste Möglichkeit Positionen von mobilen Objekten darzustellen, ist die Darstellung als Punkt (konstante Funktion), also die Annahme, dass das Objekt seine Position beibehält. Hier wird die zuletzt indexierte Position so lange als gültig angenommen, bis sich das Objekt um mehr als eine bestimmte Strecke x_{dif} von dieser entfernt hat. Erst dann schickt der Client ein Update mit der aktuellen Ortsangabe zum Server.

Abbildung 7a veranschaulicht dies. Sie zeigt die Bewegung eines Objekts, welches mit Position P1 indexiert ist. Ein Update erfolgt erst bei Position P2, wenn sich das Objekt weiter als x_{dif} von Punkt P1 entfernt hat (Verlassen des Kreises). Vor dem Update gibt der Index immer P1 als aktuelle Objektposition zurück.

Die vom BB^x-Index verwendete Möglichkeit zur Darstellung der vorhergesagten Objektpositionen ist die Vektordarstellung. Hierbei gibt Vektor $\overrightarrow{x_0}$ die Position zu einem bestimmten Zeitpunkt t_0 an, Vektor $\overrightarrow{v}$ Richtung und Geschwindigkeit der Bewegung. Daraus ergibt sich eine lineare Funktion (Funktion

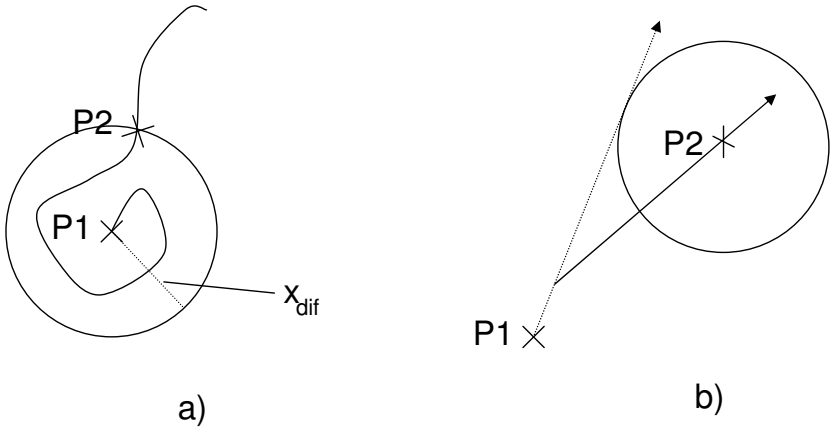

Abbildung 7. Beispiel für Punkt- und Vektordarstellung

1) über die Zeit, mit deren Hilfe man die voraussichtlichen zukünftigen Objektpositionen berechnen kann.

$$\overrightarrow{x}(t) = \overrightarrow{x_0} + \overrightarrow{v} \cdot t \tag{1}$$

Mit Funktion 1 werden nur gerade Bewegungen exakt beschrieben. Ungerade Bewegungen können nur durch lineare Bewegungen angenähert werden.

Man betrachtet hierzu das Updateverhalten im 2D-Szenario in Abbildung 7b. P1 entspricht dem Startvektor $\overrightarrow{x_0}$. Die gestrichelte Linie beschreibt die Richtung, in der sich das Objekt zum Updatezeitpunkt t_0 fortbewegt hat. Nach einem Richtungswechsel bewegt sich das Objekt allerdings entlang der durchgezogenen Linie. Bei angenommener konstanter Geschwindigkeit hat sich das Objekt an Position P2 zu weit von seiner durch Funktion 1 vorhergesagten Position entfernt, es muss ein Update erfolgen. Hierbei wird das Update aktiv vom Objekt ausgelöst. Dazu muss dieses sowohl über seine aktuelle Position informiert sein, als auch über die durch Funktion 1 vorhergesagte Position und die Genauigkeit, mit welcher der Index arbeitet.

Die Repräsentation von Objektpositionen als lineare Funktion bringen dem BB^x-Index mehrere Vorteile. Lineare Funktionen sagen zukünftige Positionen häufig besser voraus als konstante Funktionen. Wie Experimente ergeben haben, lässt sich im Fall eines Autos die Updatehäufigkeit dadurch durchschnittlich um ein Drittel reduzieren [6]. Desweiteren lassen sie sich kompakt darstellen und durch die eingesparten Updates verringert sich die Kommunikation zwischen Server und Client, was positive Auswirkung sowohl auf die Netzleistung, als auch auf den Energieverbrauch der mobilen Geräte hat.

Es gibt Szenarien in denen lange Zeit oder sogar niemals ein Update der indexierten Objektposition notwendig wäre. Naheliegendes Beispiel für die konstante Funktion wäre ein parkendes Auto. Während der Parkzeit müsste die Position im Index nie verändert werden. Äquivalent könnte bei der Vektordarstellung ein

Objekt lange Zeit seine Richtung und Geschwindigkeit nicht ändern, wie etwa ein Flugzeug.

Beim BB^x-Index wird in so einem Fall nach Δt_{mu} ein Update erzwungen. Dafür ist keine Kommunikation mit dem Client notwendig. Die neue Position wird mit Funktion 1 berechnet und muss somit nicht mit der tatsächlichen Position übereinstimmen.

Die Zeitintervalle der Länge Δt_{mu} können noch mal in n gleiche Stücke partitioniert werden (s. Abbildung 8 mit $n = 2$). Diese werden dann Phasen genannt. Mit jeder Phase wird ein Zeitpunkt t_{lab} assoziiert, der dem Endzeitpunkt der Phase entspricht. Für jede Phase wird ein B^+-Baum angelegt.

Der BB^x-Index ist also ein Wald von B^+-Bäumen und Δt_{mu} hilft uns, die Gültigkeitsdauer der Bäume zu beschränken. Die erzwungenen Updates führen zwar zu einem Overhead, aber die dadurch eingeschränkte Gültigkeitsdauer der Bäume führt auch zu einer größeren Effizienz bei der Bearbeitung von spatiotemporalen Anfragen.

Ein größeres n führt so zu einem höheren Speicherbedarf des BB^x-Index, aber führt wie auch ein kleineres Δt_{mu} dazu, dass Anfragen effizienter bearbeitet werden können. Mit $n = 2$ scheint für die Praxis ein guter Kompromiss gefunden zu sein [3].

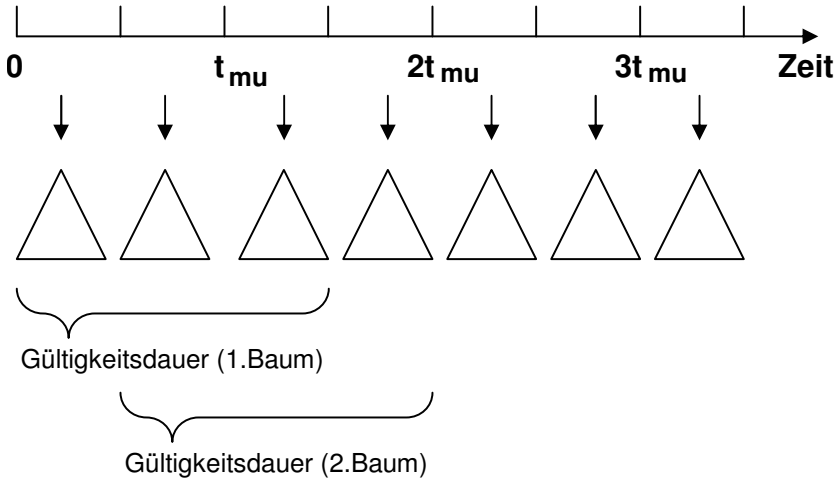

Abbildung 8. BB^x-Index mit $n = 2$

Der B^+-Baum ist eine Erweiterung des eindimensionalen B-Baums. Ein wesentlicher Unterschied ist, dass er Datenelemente nur in den Blättern speichert. Dies führt zu einem höheren Verzweigungsgrad und einer geringeren Baumhöhe. Die Motivation, im BB^x-Index den B^+-Baum zu verwenden, liegt an dessen großer Leistungsfähigkeit. Er ist sehr effizient sowohl bei Updates, als auch bei Anfragen. Des Weiteren wird er schon in zahlreichen Datenbanksystemen ver-

wendet, was eine Integration des BB^x-Index in bestehende Datenbanksysteme erleichtert.

Problem bei der Verwendung des B^+-Baums ist jedoch, dass er eine eindimensionale Indexstruktur darstellt. Positionsangaben sind mehrdimensional. Will man den B^+-Baum zum Indexieren von Ortsangaben verwenden, so müssen die mehrdimensionalen Positionsangaben linearisiert werden. Lin et al. [1] verwenden dazu eine *raumfüllende Kurve*.

Eine raumfüllende Kurve ist eine eindimensonale Linie. Sie durchläuft den n-dimensionalen Raum, in dem sich die Objekte bewegen, so, dass sie sich an jedem Punkt genau ein Mal befindet. Gute raumfüllende Kurven haben die Eigenschaft, die Ordnung zwischen Objektpositionen zu bewahren. Das bedeutet, dass Objekte, die im höherdimensionalen Raum nahe beieinander liegen, dies auch im eindimensionalen Raum tun. Das x im BB^x-Index steht für eine beliebige raumfüllende Kurve.

Beispielhaft für eine raumfüllende Kurve ist in Abbildung 9 die Arbeitsweise der Hilbertkurve beschrieben. Sie würde in diesem zweidimensionalen Fall zum Beispiel Position (3,4) auf den Wert 11 abbilden. 126 114 372 358 Die

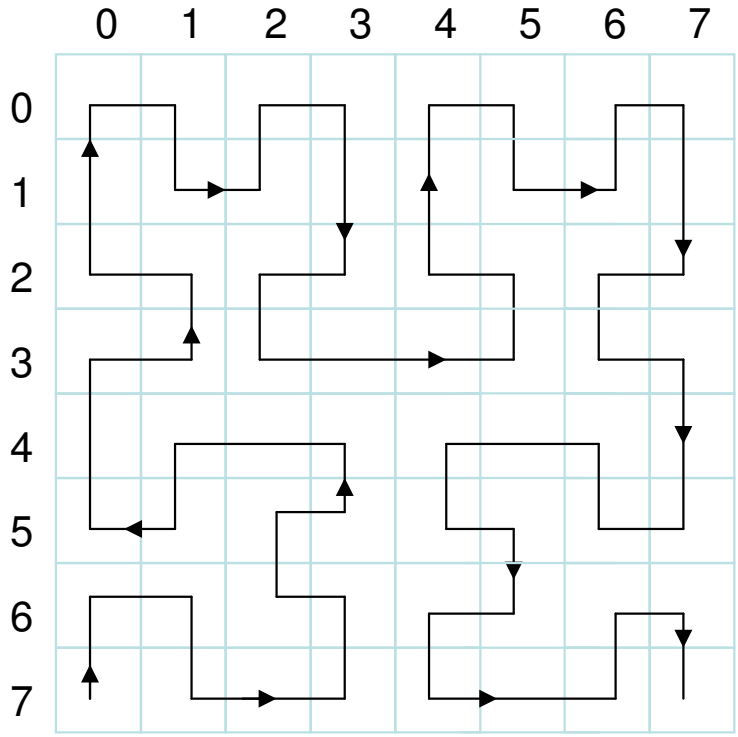

Abbildung 9. Hilbert-Kurve

Speicherung der Objektpositionen über die Zeit funktioniert im BB^x-Index nun folgendermaßen. Sendet ein Client eine Positionsänderung an den Server, so ermittelt dieser anhand des Updatezeitpunkts t_u die Phase bzw. den B^+-Baum, in dem das Objekt indexiert werden muss. Mit dem assoziierten t_{lab} wird dann mit folgender Formel

$$x(t_{lab}) = x_0 + v \cdot (t_{lab} - t_u) \tag{2}$$

die Position ausgerechnet, die das Objekt zum Zeitpunkt t_{lab} haben würde, falls es sich in gleicher Richtung mit gleicher Geschwindigkeit fortbewegen würde. Auf diese mehrdimensionale Positionsangabe wird die raumfüllende Kurve angewandt und man erhält einen Wert x_{rep}, den wir als Schlüsselwert im B^+-Baum verwenden. Die im Baum indizierten Objektpositionen beziehen sich also alle auf den gleichen Zeitpunkt t_{lab}. Dadurch erreicht man, dass die durch die raumfüllende Kurve herbeigeführte räumliche Ordnung nicht verloren geht.

Zum Schlüsselwert x_{rep} kommt ein Zeiger auf das Objekt hinzu, sowie eine Anfangs- und Endzeit. Die Anfangszeit ist der Zeitpunkt an dem das Objekt in den Baum eingefügt wird, die Endzeit entspricht dem Zeitpunkt an dem ein neues Update erfolgt ist, also wenn die Abweichung von der tatsächlichen Position zu der mit Funktion (1) errechneten Position, größer als x_{dif} ist (s. Abbildung 7b) oder wenn nach Δt_{mu} ein Update vom System erzwungen wird.

Dank Δt_{mu} hat ein B^+-Baum im BB^x-Index also eine maximale Gültigkeitsdauer vom Phasenanfang bis Phasenende+Δt_{mu}, was für $n = 2$ eine maximale Gültigkeitsdauer von 1,5Δt_{mu} bedeutet. Dies ergibt sich dadurch, dass eine Objektbewegung bis zum Phasenende+Δt_{mu} gültig ist, wenn das Objekt genau zum Zeitpunkt t_{lab} in den Baum eingefügt wurde und die indexierte Position die maximale Zeit Δt_{mu} gültig ist. In Abbildung 8 wird ein Objekt, das zum Zeitpunkt $0,5t_{mu}$ ein Update schickt, in den ersten Baum gespeichert. Baum 1 hat also eine Gültigkeitsdauer von $0-1,5t_{mu}$, Baum 2 entsprechen eine Gültigkeitsdauer von $0,5t_{mu} - 2t_{mu}$ usw.

Durch seine Darstellung der Objektpositionen als lineare Funktionen über der Zeit reduziert der BB^x-Index die Updatehäufigkeit für Objekte, die vergleichsweise selten ihre Richtung oder Geschwindigkeit ändern (z. B. Kraftfahrzeuge). Für Objekte, welche ständig ihre Richtung und Geschwindigkeit ändern, bringt diese Methode allerdings keinen Vorteil. In besonderem Maße positiv auf die Performanz wirkt sich der B^+-Baum aus. Dieser skaliert gut mit der Anzahl der Einträge und ermöglicht bei vielen Updates und gleichzeitigen Anfragen eine effiziente Anfragebearbeitung.

Bei Anfragen auf diese Indexstruktur ist zu beachten, dass erhaltene Positionsänderungen mit Hilfe von Funktion 1 immer zum Phasenendzeitpunkt gespeichert werden. Anfragen können jedoch zu jedem beliebigen Zeitpunkt gestellt werden. Die Positionen zum Anfragezeitpunkt stimmen natürlich nicht mit den indexierten Positionen überein, da sich die Objekte in diesem Zeitraum bewegt haben. Bei räumlichen Anfragen muss deshalb, abhängig von der maximalen Ausbreitungsgeschwindigkeit der Objekte und der Zeitdifferenz zwischen dem Anfragezeitpunkt und t_{lab}, das Anfragefenster erweitert werden, um somit alle potentiellen Objekte zu erhalten, welche die Anfrage erfüllen könnten. Aus

dieser Objektmenge wird dann das tatsächliche Ergebnis herausgefiltert. Details dazu finden sich in [3].

3.2 Reduzierung der Updatekosten

Ein Index, der im Gegensatz zum BB^x-Index nur aktuelle Objektpositionen indexiert, ist der *Lazy-Update Grid-based Index (LUGrid)* [2]. Mit ihm können nur räumlich beschränkte Anfragen beantwortet werden.

Wie schon erwähnt, eignen sich die traditionellen mehrdimensionalen Datenbankindizes nicht für die hohen Updateraten mobiler Objekte. Vor einem Update muss zuerst der alte Indexeintrag gefunden und gelöscht werden, bevor der neue eingefügt werden kann. Zusätzlich wird, um das Auffinden der alten Einträge zu beschleunigen, meistens ein zweiter Hilfsindex angelegt. Für diesen fallen zusätzliche Wartungsarbeiten an, wenn ein Objekt seinen Aufenthaltsort auf der Platte ändert. Alles in allem ist ein Positionsupdate aufgrund der vielen benötigten Plattenzugriffe recht teuer.

Während der BB^x-Index versucht, die Anzahl der Updates zu verringern, ist es das Ziel von LUGrid, die Kosten für ein einzelnes Update zu reduzieren und dies ohne einen zusätzlichen Hilfsindex.

Der LUGrid basiert, wie der Name schon erkennen lässt, auf der GridFile Index [4]. Das GridFile teilt einen mehrdimensionalen Datenraum in mehrere Zellen auf. Die Daten einer Grid-Zelle werden in einem sogenannten Bucket gespeichert, welcher in der Regel einer Plattenseite entspricht. Jedes Bucket hat eine bestimmte Kapazität an Datensätzen, die es aufnehmen kann. Wird die Kapazitätsgrenze überschritten, so muss der Block geteilt werden. Zur besseren Speicherbelegung kann es auch vorkommen, dass mehrere Blocks im gleichen Bucket abgespeichert sind. Das Herzstück des GridFiles ist das Grid-Verzeichnis. In diesem wird die Aufteilung der Zellen und eine Referenz zu den Buckets gespeichert. Eine mögliche Aufteilung für den zweidimensionalen Raum kann man in Abbildung 10 sehen.

LUGrid erbt vom GridFile die Struktur und die Möglichkeit sich durch Mischen und Teilen von Grid-Zellen an unterschiedliche Raumverteilungen der Objekte anzupassen. Desweiteren bedient sich LUGrid der *Lazy-insertion-* und *Lazy-deletion*-Technik.

Die meisten traditionellen Indexstrukturen für räumliche Daten aktualisieren den Index direkt nachdem sie vom mobilen Objekt ein Positionsupdate erhalten haben, führen also immer nur ein Update gleichzeitig aus. Lazy-insertion spart Ein-/Ausgabe-Operationen, indem es Updates, welche die gleiche Plattenseite betreffen, sammelt und diese gemeinsam auf die Platte ausschreibt.

Weitere Updatekosten spart LUGrid durch die Lazy-deletion-Technik. Durch sie wird bei einem Update nicht erst der alte Objekteintrag aufgesucht und entfernt, sondern direkt die neue Position eingefügt.

Um Lazy-deletion und Lazy-insertion zu ermöglichen, erweitert LUGrid das GridFile um zwei Hilfsstrukturen. Die eine ist das sogenannte *Memory Grid (MG)*. Dabei handelt es sich um das herkömmliche GridFile-Verzeichnis, welches aber beim LUGrid im Hauptspeicher und nicht auf der Platte abgelegt wird.

Das MG wird außerdem um die Fähigkeit erweitert, eine gewisse Anzahl von Objektupdates pro Grid-Zelle zu puffern.

Die andere Struktur nennt sich *Miss-Deletion Memo (MDM)*. Sie zeichnet Objekte auf, welche aufgrund der Lazy-deletion einen Löschvorgang verpasst haben. Diese befinden sich noch auf der Platte, obwohl sie bereits gelöscht sein müssten. Ihre Position ist veraltet. Neue Objektpositionen lassen sich so direkt in den Index einfügen und Löschvorgänge alter Objektpositionen können auf einen späteren Zeitpunkt verschoben werden. Die extra Ein-/Ausgabe-Operationen für das Löschen fallen somit weg. Sie werden nachgeholt, wenn sich die betroffene Seite auf Grund eines Updates im Speicher befindet. Die MDM-Struktur wird ebenso wie das MG ständig im Hauptspeicher gehalten, was aufgrund ihrer überschaubaren Größen problemlos möglich ist.

Die genaue Funktionsweise des LUGrid soll anhand des Beispiels in Abbildung 10 verdeutlicht werden. Zu sehen sind das Disk Grid (DG), das Memory Grid (MG) und das MDM, wobei das DG das herkömmliche GridFile ist, das auf der Platte gespeichert wird. Das MG kann in unserem Beispiel zwei Objektupdates pro Zelle puffern, bevor diese auf die Platte ausgeschrieben werden müssen.

Das DG besteht aus fünf Zellen(A, B, C, D, E), das MG aus sechs(1, 2, 3, 4, 5, 6). Anzumerken ist, dass jeder MG-Zelle genau eine DG-Zelle zugeordnet wird (z.B. MG-Zelle 2 zu DG-Zelle B). Der gleichen DG-Zelle können hingegen mehrere unterschiedlichen MG-Zellen zugeordnet sein (MG-Zellen 2 und 3 sind DG-Zelle B zugeordnet).

Zu Beginn sind im DG neun Objekte (O1-O9) gespeichert. Die einzigen Einträge im MG sind O1 und O9 in Zelle 2. Dies bedeutet, dass Objekte O1 und O9 dem Server eine neue Position mitgeteilt haben. Nach dem ersten Update wurde aber dank der Lazy-insertion die Positionsänderung nicht direkt ins DG geschrieben, sondern so lange gewartet, bis die Kapazitätsgrenze für die DG-Zelle erreicht wurde. Mit 2 Einträgen ist dies der Fall, d.h. der Puffer für Zelle 2 ist voll und die Positionsupdates werden auf die Platte geschrieben. Dazu wird die der MG-Zelle 2 zugeordnete DG-Zelle B in den Speicher geladen.

Aufgrund der Lazy-deletion kann es passieren, dass veraltete Objektpositionen in einer Zelle gespeichert sind. Deshalb wird nach dem Laden der DG-Zelle in den Speicher zuerst untersucht, ob für ein Objekt in der DG-Zelle ein Eintrag im MDM existiert, d.h. man prüft, ob Objekte in der DG-Zelle einen Löschvorgang verpasst haben und dieser jetzt nachgeholt werden kann.

Einträge im MDM haben die Form(OID, OLoc, MDnum). OID ist der Objektbezeichner, OLoc die aktuelle Position des Objekts und MDnum beschreibt wie oft ein Objekt einen Löschvorgang verpasst hat. Für MDM-Eintrag (O4, (12,13), 3) wüssten wir beispielsweise, dass für Objekt O4 noch drei veraltete Einträge im DG existieren und dass O4 aktuell mit Position (12,13) indexiert ist.

In unserem Beispiel ist die MDM zu Beginn leer (s. Abbildung 10.1c)) und somit ist sichergestellt, dass sich keine veralteten Einträge in DG-Zelle B befinden. Nun wird überprüft, ob sich die Objekte O1 und O9 in Zelle B befinden. Ist

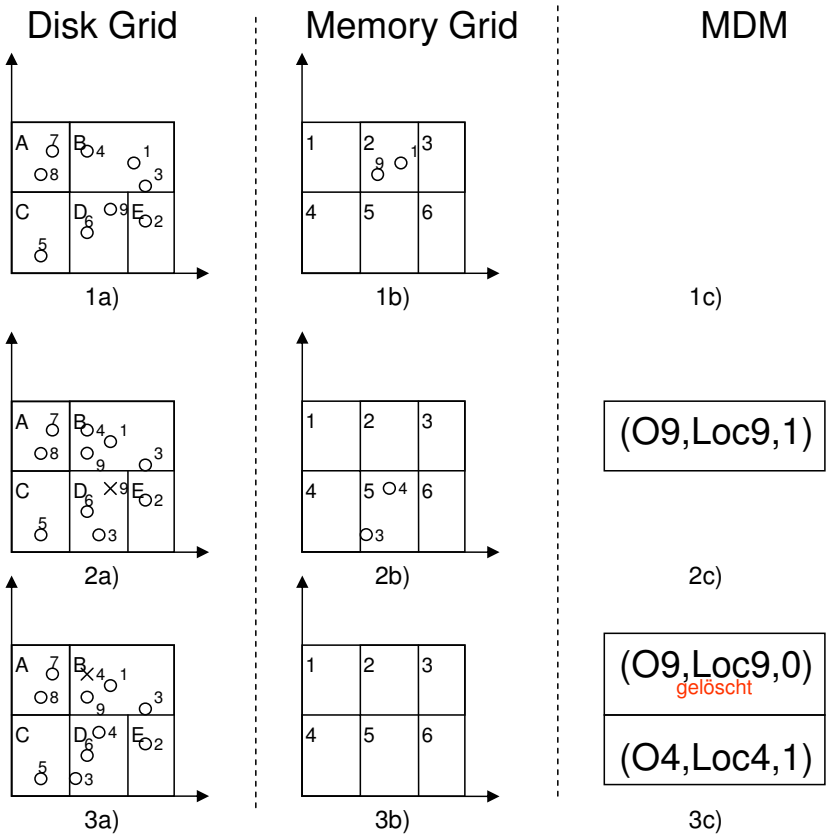

Abbildung 10. Beispiel für das LUGrid

dies der Fall (O1 in unserem Beispiel), so wird einfach die alte Position durch die neue ersetzt. Danach wird erneut nach einem MDM-Eintrag für das Objekt gesucht. Falls ein Eintrag gefunden wird, muss OLoc entsprechend aktualisiert werden. Befindet sich das Objekt nicht in der Zelle (Objekt O9 in unserem Beispiel), so wird es neu hinzugefügt. Da sich O9 nicht in Zelle B befindet, muss es mit einer veralteten Position weiterhin in einer anderen DG-Zelle gespeichert sein. Dies wird im MDM vermerkt.

Da für O9 noch kein MDM-Eintrag exisitiert, muss ein neuer angelegt werden, bei dem MDNum auf 1 gesetzt wird (s. Abbildung 10.2c)). Hätte für O9 schon ein MDM-Eintrag existiert, so hätte bei diesem OLoc auf die neue Position aktualisiert und MDnum um eins erhöht werden müssen.

Abbildung 10.2a) zeigt nun das DG nachdem die 2 Updates ins DG gespeichert wurden. Die veraltete Position von O9 (mit einem x gekennzeichnet) ist noch immer in DG-Zelle D eingetragen. Objekt O9 wurde also vor dem Einfügen nicht aus D gelöscht (Lazy-deletion) und die Updates für O1 und O9 wurden zusammen ausgeschrieben (Lazy-insertion).

In Abbildung 10.2b) sehen wir Updates von Objekt O3 und O4 in MG-Zelle 5. Erneut wird die zugehörige DG-Zelle in den Speicher geladen (hier Zelle D) und das MDM nach veralteten Einträgen überprüft. Dieses Mal finden wir mit O9 einen Eintrag, der sich sowohl in Zelle D als auch im MDM befindet. Die im MDM gespeicherte Objektposition Loc9 stimmt nicht mit der Position von O9 in DGZelle D überein. So weiß man, dass die Position von O9 in Zelle D veraltet ist. Sie wird entfernt und MDNum im MDM-Eintrag von O9 um eins reduziert. Da dies in unserem Beispiel MDNum=0 ergibt, kann der komplette Eintrag für O9 aus dem MDM gelöscht werden. Analog zu unserem vorherigen Fall, werden dann Objekte O3 und O4 in das Disk Grid eingefügt, wobei für O4 ein MDM Eintrag angelegt werden muss (s. Abbildung 10.3c)).

Anzumerken ist, dass der LUGrid Index eine Hash-Tabelle für alle OIDs anlegt, die im Memory Grid gespeichert sind. Dies ist hilfreich für den Fall, dass ein Update zum Server geschickt wird und sich ein älteres Update des gleichen Objekts noch immer im MG befindet. Hierbei wird das alte Update einfach durch das Neue ersetzt und die Hash-Tabelle angepasst.

Auf das Splitten und Mischen von Grid-Zellen sind wir in unserem Beispiel nicht eingegangen. Grundsätzlich muss bei jedem Update überprüft werden, ob eine Zelle überläuft oder unterbesetzt ist. Der erste Fall führt zu einem Splitten der Zelle. Beim zweiten Fall wird zur Effizienzsteigerung die Zelle mit einer anderen gemischt. Dies funktioniert genauso wie beim Standard-GridFile. Nähere Informationen dazu finden sich in [4].

4 Zusammenfassung

Die Indexierung von beweglichen Objekten sowie kontinuierliche räumliche Anfragen stellen traditionelle Datenbankindizes vor große Schwierigkeiten. Ziel dieser Ausarbeitung ist es Wege aufzuzeigen, die diese Schwierigkeiten so gut wie möglich beheben.

Kontinuierliche mobile Anfragen haben die Eigenschaft, dass ein Ergebnis, das gerade noch richtig war, im nächsten Moment schon wieder falsch sein kann. Eine konservative Vorgehensweise, die immer ein richtiges Ergebnis liefern würde, würde bei jedem Positionswechsel eine neue Anfrage an den Server schicken.

Als ein Verfahren, das Anfragen gegenüber dieser konservativen Vorgehensweise einspart, wurde der Ansatz von Zhang et al. [8] vorgestellt. Dazu berechnen sie außer dem Anfrageergebnis noch eine Region, in der das Ergebnis seine Gültigkeit behält. So lange sich ein Client in diesem Gültigkeitsbereich befindet, weiß er, dass sein aktuelles Ergebnis noch gültig ist. Eine neue Anfrage ist erst nach Verlassen der Gültigkeitsregion erforderlich. Das Verfahren ist sowohl für Nearest-Neighbour-Anfragen als auch für die Bereichsanfragen anwendbar, funktioniert aber nur bei statischen Objekten.

Als Beispiel für ein Verfahren, das auch Anfragen auf beweglichen Objekten unterstützt, wurde der von Mokbel und Aref entworfene *Generic and Progressive Algorithm* [10] vorgestellt.Er bildet ein Framework, mit dem sich eine Vielzahl von räumlichen Anfragearten beantworten lassen. Die eingehenden Datentupel der mobilen Objekten werden direkt im Hauptspeicher verarbeitet und müssen nicht auf einem persistenten Medium materalisiert werden. Dies führt zu einer höheren Effizienz und einer verkürzten Antwortzeit. Das Verfahren funktioniert sowohl für statische als auch für dynamische Objekte.

Mobile Objekte verursachen eine Vielzahl von Updates. Um die Kosten für ein Update zu verringern, entwickelten Xiong et al. [2] den *Lazy-Update Grid-based index*. Der Index basiert auf dem GridFile, hält das Grid-Verzeichnis aber im Hauptspeicher und erweitert es um einen Objektpuffer, in dem Updates gesammelt werden, bevor sie dann gemeinsam auf die Platte geschrieben werden (Lazy-insertion). Weitere Plattenzugriffe spart LUGrid durch seine Lazy-deletion Technik, die durch eine im Hauptspeicher befindliche Hilfsstruktur ermöglicht wird. Sie bewirkt, dass die neue Objektposition direkt in die Datenbank eingefügt werden kann, ohne die alte Position vorher gelöscht zu haben.

Lin et al. [1] können sowohl vergangene, als auch aktuelle und zukünftige Positionen von mobilen Objekten indexieren. Um dem Problem der hohen Updateraten zu begegnen, bedienen sie sich dem bekannt leistungsfähigen B^+-Baum. Um den B^+-Baum zur Indexierung von mehrdimensionalen Daten verwenden zu können, linearisieren sie diese Daten vorher mit Hilfe einer raumfüllenden Kurve. Die Anzahl der Updates reduzieren sie dadurch, dass sie Objektpositionen als lineare Funktion über der Zeit interpretieren. Die vergangenen Objektpositionen indexieren sie, indem sie Zeitphasen definieren und für jede dieser Phase einen B^+-Baum anlegen, der die Positionen über diesen Zeitraum abspeichert.

Die technischen Möglichkeiten für neue ortsabhängige Anwendungen sind also gegeben. Bleibt abzuwarten, wann konkrete Ausprägungen auf den Markt gebracht werden und ob die Benutzerfreundlichkeit und der praktische Nutzen ausreichen werden, um eine große Benutzermenge zufriedenstellen zu können.

Literatur

1. D.Lin, C.S. Jensen, B.C. Ooi, and S. Saltenis. Efficient Indexing of the Historical, Present, and Future Positions of Moving Objects. Mobile Data Management 2005: 59-66.
2. X.Xiong, M.F. Mokbel, and W.G.Aref. LUGrid: Update-tolerant Grid-based Indexing for Moving Objects. MDM 2006: 13.
3. C.S. Jensen ,D.Lin, and B.C. Ooi. Query and Update Efficient B^+-Tree Based Indexing of Moving Objects. Proc. VLDB 2004: 768-779.
4. J.Nievergelt, H. Hinterberger, and K. C. Sevcik. The Grid File: An adaptable, Symmetric Multikey File Structure. TODS, 9(1), 1984.
5. A. Guttman. R-Trees: A Dynamic Index Structure for Spatial Searching. In SIGMOD, 1984.
6. A. Civilis, C. S. Jensen, J. Nenortaite, and S. Pakalnis. Efficient Tracking of Moving Objects with Precision Guarantees. In Proc. MobiQuitious,pp. 164-173, 2004.
7. Dey, A.K. and Abowd, G.D. Towards a better understanding of context and context-aware applications. In Workshop on Software Engineering for Wearable and Pervasive Computing, 2000.
8. J. Zhang, M. Zhu, D. Papadias, Y. Tao, and D. L. Lee. Location-based Spatial Queries SIGMOD Conference 2003: 443-454
9. Y. Tao, D.Papadias. Time Parameterized Queries in Spatio-Temporal Databases. In SIGMOD, 2002.
10. M. F. Mokbel, and W. G. Aref. GPAC: Generic and Progressive Progressing of Mobile Queries over Mobile Data. MDM, 2005.