Moving Objects – Bewegliche Objekte

Martin Tritschler

Technische Universität Kaiserslautern

Abstract. *Diese Arbeit im Rahmen des Seminars "Mobile and Context-aware Database Technologies and Applications", betreut von Prof. Dr.-Ing. Dr. h. c. Theo Härder, beschäftigt sich mit der Verwaltung von Positionsangaben beweglicher Objekte („Moving Objects") in Datenbanksystemen. Dabei treten Anforderungen und Probleme auf, die es bei der Verwaltung statischer Objekte nicht gibt. In dieser Ausarbeitung werden Lösungsansätze vorgestellt, um die Position beweglicher Objekte zu modellieren, Anfragen mit zeitlichem Bezug auf ihnen zu bearbeiten und die Problematik der Ungenauigkeit zu behandeln. Weiterhin werden spezielle Indexstrukturen vorgestellt, die trotz der hohen Aktualisierungsraten effizient die Positionen der Objekte indexieren können.*

1 Einleitung

Dank des enormen Fortschritts in der Computertechnologie und der gestiegenen Verfügbarkeit drahtloser Netzwerke genießen kleine mobile Computer, wie PDAs, „Subnotebooks" und „Smartphones", in den letzten Jahren eine immer größere Verbreitung in der Bevölkerung. Beschränkte sich der Funktionsumfang von Mobiltelefonen in der Vergangenheit noch auf das reine Telefonieren, so steht mittlerweile eine große Anzahl von weiteren Kommunikationsdiensten zur Verfügung, wie beispielsweise mobiler Internetzugang. Hierbei zeigt sich in letzter Zeit vor allem ein Trend zu kontextbewussten Anwendungen, also Anwendungen die den Kontext, in dem sich das System befindet, berücksichtigen. Der Begriff „Kontext" wird in der Literatur auf verschiedene Arten definiert, wobei eine für das „Mobile Computing" passende Definition in diesem Zusammenhang von Dey [1] geliefert wird. Demnach ist Kontext definiert als „jegliche Informationen, die benutzt werden können, um die Situation von Personen, Orten oder Objekten zu charakterisieren, die relevant für die Interaktion zwischen einem Benutzer und einer Applikation sind, einschließlich dem Benutzer und der Applikation selbst". Die derzeit am meisten genutzten Kontextinformationen stellen Ortsinformationen dar, mit denen Benutzern Dienste angeboten werden können, die auf ihre aktuelle Position zugeschnitten sind. Mit Mobiltelefonen lassen sich mittlerweile ortsbezogene Informationen anfordern, wie etwa die sich in der Nähe des Benutzers befindlichen Hotels oder Restaurants. Den bekanntesten Anwendungsbereich für solche Ortsinformationen stellen momentan Navigationssysteme dar. Die in einem Navigationssystem befindliche Datenbasis hat hierbei die Eigenschaft, sich nur relativ selten zu verändern. Orte oder Länder ändern nur sehr selten ihre Position und, falls Straßen neu gebaut oder verlegt werden, wird in einem Navigationssystem eine solche

Änderung nicht als kontinuierlicher Prozess, sondern als diskrete Änderung wiedergegeben. Möchte man hingegen die Positionen sich permanent bewegender Objekte wie Fahrzeugen angeben, ergeben sich für die zugrunde liegenden Datenbanksysteme ganz neue Anforderungen. Solche Objekte mit ständig wechselnder Position werden bewegliche Objekte, im Englischen „Moving Objects, genannt. In dieser Ausarbeitung werden die mit der Verwaltung beweglicher Objekte verbundenen Probleme aufgezeigt und aktuelle Lösungsansätze vorgestellt.

Ein Datenbanksystem zur Verwaltung der Positionen beweglicher Objekte wird „Moving-Objects-Datenbanksystem“ (kurz MOD-System) genannt. Die in einem solchen System verwalteten beweglichen Objekte können hierbei beliebiger Art sein. Beispielsweise sind Fahrzeuge, Flugzeuge oder auch Personen möglich, die mit mobilen Computersystemen ausgerüstet sind, welche eine Positionsbestimmung und Übermittlung an das MOD-System ermöglichen. Die Objekte ermitteln dabei oft selbstständig, z.B. per GPS[1], ihre Position und transferieren diese durch beliebige Funknetzwerke wie Wireless-LAN, GSM und UMTS an das MOD-System. Es sind aber auch andere technische Ansätze zur Bestimmung der Position und deren Übertragung möglich. Ein Überblick über eingesetzte Techniken ist in [1] zu finden.

Der weitere Verlauf dieser Ausarbeitung ist wie folgt gegliedert: In Kapitel 2 werden die speziellen Anforderungen an ein MOD-System aufgezeigt und die damit verbundenen Probleme erläutert. In den darauf folgenden Kapiteln werden dann einige ausgesuchte Lösungsansätze für diese Problemstellungen dargestellt. Kapitel 3 befasst sich mit der Modellierung der Position, der Formulierung und Auswertung von Anfragen und dem Umgang mit der Ungenauigkeit in MOD-Systemen. In Kapitel 4 werden einige zum Einsatz kommende Indexstrukturen erläutert. Kapitel 5 wird mit einer Zusammenfassung und einem Ausblick abschließen.

2 Anforderungen an Systeme zur Verwaltung beweglicher Objekte

Es gibt zwei verschiedene Ansätze, die sich mit beweglichen Objekten beschäftigen. Die so genannte „Location-Management-Perspektive“ ist daran interessiert, die aktuellen Positionen von beweglichen Objekten zu erfassen, ihre zukünftigen Lokationen zu extrapolieren und Anfragen auf ihnen zu beantworten. Die Verwaltung einer großen Taxi-Flotte oder einer Spedition sowie die Überwachung des Luftraums sind Anwendungsgebiete dieses Ansatzes. Auftretende Anfragen sind beispielsweise „finde alle Taxen, die sich gegenwärtig im 1km-Radius um die Gottlieb-Daimler-Straße 10 befinden“ (Q1) oder „finde alle Flugzeuge, die in den nächsten 20 Minuten das Bundesgebiet überfliegen werden“ (Q2). Die Verwaltung vergangener Objektpositionen ist hierbei nicht von Bedeutung. Dieser Sicht gegenüber steht die „spatio-temporale Perspektive“. Hier geht es darum, wie sich Objektpositionen im Zeitverlauf verändert haben. Nicht nur der gegenwärtige Zustand eines Objekts ist also von Bedeutung, sondern vor allem auch seine komplette Vergangenheit und die Beziehungen zu anderen Objekten im Zeitverlauf. Beispielhafte Anfragen sind hierbei

[1] Oder vielleicht auch einmal durch das Galileo-System.

„zeige den Verlauf der Nullgradgrenze in Deutschland am 18.01. um 4 Uhr“ (Q3) oder „zeige die Route, die Taxi 22 gestern zwischen 16 und 19 Uhr zurückgelegt hat“ (Q4).

Was muss ein Datenbanksystem bereitstellen, um bewegliche Objekte verwalten zu können? Zieht man die erwähnte Anfrage Q2 heran, so beinhaltet sie als räumliche Komponente das Bundesgebiet, beispielsweise dargestellt (und angenähert) durch ein Polygon, als zeitliche Komponente den Zeitraum von jetzt bis 20 Minuten in der Zukunft. Dazu kommt noch eine ggf. sehr große Zahl sich schnell bewegender Objekte, den Flugzeugen. Zur Verwaltung der Lokationen werden bestimmte räumliche Datentypen wie Punkte und Linien, aber auch räumlich ausgedehnte Typen wie Flächen und Polygone oder, je nach Anwendungsgebiet, auch dreidimensionale Körper benötigt. Des Weiteren müssen Funktionen oder Methoden vorhanden sein, um grundlegende geometrische Anfragen wie Abstände, Schnittpunkte und Schnittflächen realisieren zu können. Oft benutzte Anfragen wie die „Nearest-Neighbour-Anfrage“ („finde die drei nächst gelegenen Objekte des Typs x“) oder die Bereichsanfrage („finde alle Objekte vom Typ x im Umkreis von 3 km“) sollten auch gegeben sein. Außerdem werden adäquate Index- und Datenstrukturen benötigt, um die räumlichen Informationen effektiv verarbeiten zu können. In so genannten räumlichen Datenbanken[2] wurden solche Anforderungen schon umgesetzt, sie kommen heute hauptsächlich in Geo-Informationssystemen zur Anwendung. Weitergehend muss es auch möglich sein, Anfragen mit zeitlichem Bezug zu verarbeiten. Möchte man in einer firmeninternen Personaltabelle nicht nur aktuelle, sondern auch ehemalige Mitarbeiter mitsamt ihrer Arbeitszeit in der Firma speichern, so kann man dies durch zwei Datumsattribute leicht bewerkstelligen, muss sich aber in der Anwendungsschicht um die Verwaltung der zeitlichen Daten kümmern und dies bei Anfragen in die Anfragebedingungen mitaufnehmen. Es gibt jedoch auch Datenbanksysteme, so genannte temporale Datenbanken, welche eine integrierte Zeitverwaltung auf tieferer Ebene bereitstellen und die entsprechenden Datenstrukturen und Sprachkonstrukte besitzen, um zeitliche Anfragen sehr effizient erledigen zu können.

In einem MOD-System müssen beide oben genannte Funktionalitäten gleichzeitig vorhanden sein und es muss eben noch eine größere Anzahl beweglicher Objekte in der Datenbank verwaltet werden. Durch die permanente Bewegung treten dabei einige Anforderungen auf, die es bei der Verwaltung traditioneller, statischer Daten in dieser Form nicht gibt. In Anlehnung an [3] und [12] sollte ein Datenbanksystem für bewegliche Objekte daher für die folgenden Problemfelder Lösungen bereitstellen:

Modellierung der Position
Herkömmliche Datenbanken sind darauf ausgelegt, dass sich Daten nur ändern, wenn dies explizit gewünscht ist, sie also aktualisiert werden. Hat ein Attribut in einer Gehaltsdatenbank den Wert 50000, so soll dieser konstante Wert bei Anfragen immer zurückgegeben werden, bis er explizit verändert wird. Die Repräsentation beweglicher Objekte und Anfragen bezüglich ihrer Position stellen deshalb ein

[2] „Spatial Database“ im Englischen.

Problem für herkömmliche Datenbanksysteme dar. Um die sich permanent ändernden Lokationen zu approximieren, müssten diese ständig aktiv aktualisiert werden. Bei einer großen Anzahl zu überwachender Objekte würden dadurch auf Seite der Datenbank unzählige Transaktionen nötig werden, auf Seite der beweglichen Objekte, sofern diese ihre Position selbst z.B. über ein Funknetzwerk übermitteln, fielen große Datenmengen und eine hohe Belastung des Netzwerks an. Hierbei muss auch der Fall berücksichtigt werden, dass ein Objekt temporär nicht in der Lage ist, seinen aktuellen Standort an die Datenbank zu übermitteln, obwohl es sich weiter bewegt. Man denke dabei an Tunnel oder „Funklöcher“. Verringert man die Häufigkeit der Aktualisierungen, so muss eine steigende Ungenauigkeit der Daten in Kauf genommen werden, was auch eine unbefriedigende Lösung darstellt.

Anfragesprachen
Wie eingehend erwähnt, beinhalten Anfragen in MOD-Systemen räumliche Objekte wie Punkte, Linien oder Polygone und gleichzeitig auch zeitliche Bedingungen. In traditionellen Anfragesprachen wie SQL lassen sich solche „spatio-temporalen“ Anfragen nur sehr umständlich stellen. Es wurden zwar Anfragesprachen sowohl für zeitliche als auch für räumliche Anfragen entwickelt, dies jedoch meist getrennt voneinander. Räumliche Datenbanksysteme stellen zwar oft räumliche Datentypen und Anfrageprädikate auf ihnen bereit, jedoch sind auch hier zeitliche Bedingungen umständlich zu modellieren. Außerdem sind die dort zugrunde gelegten Datenstrukturen nicht unbedingt zur Modellierung der veränderlichen Lage von beweglichen Objekten geeignet.

Ungenauigkeit
Eine gewisse Ungenauigkeit in der Position eines beweglichen Objekts ist inhärent, egal wie oft die in der Datenbank gespeicherte Lokation aktualisiert wird. Der Zustand in der Datenbank wird sich immer zu einem gewissen Maß von dem wirklichen unterscheiden. Diese inhärente Ungenauigkeit hat einige Implikationen auf Anfragen und Datenmodellierung. Man kann bei einer Anfrage zwei Gruppen von Objekten unterscheiden: solche, die die Anfrage mit Sicherheit und solche, die sie möglicherweise erfüllen. Es stellen sich dabei die Fragen, wie man diese Ungenauigkeit quantifizieren kann und wie ein Kompromiss zwischen Ungenauigkeit und Anfragehäufigkeit zu erreichen ist. Auch muss hier in Betracht gezogen werden, dass ein Objekt zeitweise nicht in der Lage ist, seinen Standort an die Datenbank zu übermitteln.

Indexstrukturen
Um effizient die Positionen beweglicher Objekte abfragen zu können, muss es eine räumliche Indexierung geben. Die permanente Bewegung der Objekte impliziert jedoch eine dauernde Aktualisierung der Indexstrukturen, was vor allem bei einer großen Anzahl zu überwachender Objekte unakzeptabel ist. Im Datenbanksystem muss bei jeder einzelnen Aktualisierung die Transaktionskonsistenz beispielsweise durch Sperrverfahren gewahrt werden und zusätzlich noch „Logging“ für den Fehlerfall durchgeführt werden, was zu einer hohen Belastung des Systems führt. Werden dazu noch zeitliche Komponenten, wie die Position zu vergangenen oder

zukünftigen Zeiten, in die Indexstrukturen mit aufgenommen, erschwert dies weiter eine performante Verarbeitung und es stellt sich die Frage der effizienten Verwaltung dieser Datenstrukturen.

In den folgenden Kapiteln werden nun einige ausgesuchte Lösungsansätze für diese Problemstellungen erläutert.

3 Anfragen, Ungenauigkeit und die Modellierung der Position

In Kapitel 3 wird besprochen, wie die in Abschnitt 2 genannten Probleme der Modellierung, Anfrage und Unsicherheit gelöst werden können, wobei die „Location-Management-Perspektive“ zu Grunde gelegt wird, um die Problematik der sich ständig ändernden Objektpositionen hervorzuheben. Dabei wird hauptsächlich auf die von Wolf et al. in [11] und [12] vorgestellte „Database for Moving Objects“ (kurz DOMINO) und die dort verwendeten Methoden eingegangen.

3.1 Modellierung der Position

Sich dauernd verändernde Objektpositionen in einer Datenbank zu speichern, stellt in herkömmlichen Systemen ein Problem dar, weil diese darauf ausgelegt sind, dass sich Daten nur bei einer Aktualisierung verändern. Eine Lösung dieses Problems mit Verzicht auf permanente Aktualisierungen ist, nicht die Position eines Objekts zu speichern, sondern dessen Bewegungsvektor. Wird dabei eine lineare Funktion der Zeit eingesetzt, ändert sich die in der Datenbank gespeicherte Lokation ständig, ohne dass eine explizite Aktualisierung nötig ist. Nur im Falle einer Richtungs- oder Geschwindigkeitsänderung müssen die Daten explizit verändert werden, was jedoch viel seltener verkommt als bei einer Speicherung der eigentlichen Position des Objekts.

Im Rahmen des DOMINO-Projekts wurde das „Moving Objects Spatio-Temporal Model“ (kurz MOST) eingeführt. In ihm werden Bewegungsvektoren nicht als explizit sichtbarer eigener Datentyp dargestellt, sondern sie werden mit dem Konzept des dynamischen Attributs realisiert. Ein dynamisches Attribut verhält sich wie ein statisches, in diesem Sinne „normales“ Attribut, mit dem Unterschied, dass es seinen Wert selbstständig als Funktion der Zeit ändert, ohne dass ein Eingriff von außen notwendig ist. Zwei Anfragen an das gleiche dynamische Attribut, jedoch mit verschiedenen Zeitwerten, können somit unterschiedliche Ergebnisse zurückliefern. Mit dem Vorhandensein solcher Attribute sind in der Datenbank somit nicht nur Informationen über gegenwärtige, sondern auch über zukünftige Zustände eines Attributs vorhanden.

Formal lässt sich ein dynamisches Attribut wie folgt definieren [3], [11]:

Ein dynamisches Attribut *A* vom Typ *T* wird repräsentiert durch drei Subattribute *A.updatevalue*, *A.updatetime* und *A.function*, wobei *A.value* ein Attribut vom Typ *T*

ist, *A.updatetime* ein Zeitwertstempel und *A.function* eine Funktion $f: N \rightarrow T$, für die gilt $f(0)=0$. Der Wert eines dynamischen Attributs ist wie folgt definiert: Zum Zeitpunkt *A.updatetime* hat A den Wert *A.updatevalue*, welcher vom Objekt an die Datenbank übermittelt wird. Bis zur nächsten Aktualisierung von *A* ist der Wert zum Zeitpunkt *A.updatetime* + *t* als *A.updatevalue* + *A.function(t)* gegeben. Eine explizite Aktualisierung von *A* kann die Subattribute *A.value* und *A.function* ändern. Bei einem beweglichen Objekt im zweidimensionalen Raum kann man nun sein Positionsattribut *pos* durch zwei dynamische Attribute für die X- und Y-Koordinaten *pos.x* und *pos.y* repräsentieren, jedes mit eigenem *updatevalue* und eigener *updatefunction*. Explizit aktualisiert werden die dynamischen Attribute, wenn das Objekt seine Richtung oder Geschwindigkeit ändert. Aus Gründen der Einfachheit wird hier davon ausgegangen, dass der Zeitpunkt *updatetime*, an dem das Objekt mit der Datenbank kommuniziert, auch der Zeitpunkt ist, an dem die Werte in die Datenbank geschrieben werden.

Ist das Objekt ein Flugzeug, welches über einen längeren Zeitraum eine konstante Richtung und Geschwindigkeit hat, ist dieser Ansatz mit wenigen Aktualisierungen verbunden und die Datenbank kann jederzeit aktuelle und zukünftige Positionen angeben. Bei einem Auto, welches eine Route auf Straßen abfährt, müssten durch die Kurven trotzdem noch sehr viele Aktualisierungen bewerkstelligt werden. Um dem entgegenzuwirken, wurde das Konzept in [3] dahingehend erweitert, dass das Lokationsattribut sechs Subattribute *loc.route*, *loc.startlocation*, *loc.starttime*, *loc.direction*, *loc.speed* und *loc.uncertainity* besitzt. *Loc.route* ist dabei (ein Zeiger auf) ein räumliches Objekt, welches die Geometrie der Route beschreibt, auf der sich das bewegliche Objekt befindet. *Loc.startlocation* ist der Punkt auf *loc.route*, an dem sich das Objekt zum Zeitpunkt *loc.starttime* befindet. *Loc.direction* ist die relative Bewegungsrichtung, auf der sich das Objekt entlang der Route fortbewegt. *Loc.speed* ist eine Funktion *f* der Zeit, welche den Abstand auf der Route vom Punkt *loc.startlocation* zum Zeitpunkt *t* angibt, wobei gilt, $f(0)=0$. Im einfachsten Fall enthält die Funktion eine Konstante *v*, welche die Geschwindigkeit des Objekts ist. Dann ist der Abstand vom Punkt *loc.startlocation* zum Zeitpunkt *loc.starttime* + *t* folglich $v \cdot t$. *Loc.uncertainity* gibt an, wie groß die Unsicherheit der Lage des Objekts ist. Immer wenn die Unsicherheit einen bestimmten Schwellenwert überschreitet, wird das Objekt eine Positionsaktualisierung zur Datenbank senden. Auf diesen Mechanismus wird in Abschnitt 3.3 näher eingegangen. Das dynamische Attribut *loc* stellt somit eine zeitabhängige Position *(x,y)* auf der Route dar, nämlich zum Zeitpunkt *loc.starttime* + *t* die Stelle auf der Route *loc.route*, die $loc.speed \cdot t$ Entfernungseinheiten vom Punkt *loc.startlocation* in Richtung *loc.direction* entfernt ist und sich mit einer Abweichung von maximal *loc.uncertainity* weiter vorne oder hinten auf der Route befindet.

Um die Semantik einer möglichen Anfrage im MOST-Datenmodell besser zu verstehen, werden nun die theoretischen Konzepte des Datenbankzustands und der Datenbankhistorie eingeführt [3]. Ein Datenbankzustand ist eine Abbildung, die jeder Objektklasse eine Menge von Objekten des zugehörigen Typs und jedem Zeitstempel die aktuelle Zeit zuordnet. Eine Datenbankhistorie ist eine (theoretische und nicht real im System existierende) unendliche Sequenz von Datenbankzuständen, welche für jede Zeiteinheit einen Datenbankzustand mit ansteigendem Zeitstempel besitzt. Sie

startet an einem bestimmten Zeitpunkt und setzt sich unendlich weit in die Zukunft fort. Ein Attribut kann in zwei aufeinander folgenden Datenbankzuständen unterschiedliche Werte besitzen. Dies rührt daher, dass es entweder explizit aktualisiert wurde oder ein dynamisches Attribut ist, das seinen Wert selbst verändert hat. Eine Anfrage an die Datenbank im MOST-Modell ist nun nicht an einen einzelnen Datenbankzustand gerichtet, sondern sie ist als Prädikat über Datenbankhistorien anzusehen. Die Antwort einer Anfrage ist dann die Menge der erfüllenden Instanzen der Prädikatsvariablen.

Hierbei lassen sich drei verschiedene Arten von theoretisch möglichen Anfragen unterscheiden: Moment-Anfragen[3], kontinuierliche Anfragen und persistente Anfragen, wobei jede Anfrage als einer der drei Typen gestellt werden kann und dabei auch unterschiedliche Ergebnisse zurückliefert.

Moment-Anfragen
Eine Moment-Anfrage zum Zeitpunkt *t* ist eine Anfrage an die unendliche Datenbankhistorie beginnend beim Zeitpunkt *t*. Ein Autofahrer, welcher ein Navigationssystem mit Anschluss an ein MOD-System besitzt, könnte die folgenden Anfragen stellen. Bei der Anfrage „finde alle Hotels im Umkreis von 3km“ ist *t* der aktuelle Zeitpunkt, aber es kann auch ein beliebiger zukünftiger Zeitpunkt sein, wie bei „finde alle Hotels, die von meinem Standort in 20 Minuten weniger als 3km entfernt sind“. Durch „finde alle Hotels, an denen ich in den nächsten 20 Minuten mit weniger als 3 km Abstand vorbei kommen werde“ wird verdeutlicht, dass auch die Menge aller Zeitpunkte zwischen dem aktuellen und einem bestimmten zukünftigen Zeitpunkt herangezogen werden kann.

Kontinuierliche Anfragen
Eine kontinuierliche Anfrage zum Zeitpunkt *t* ist eine Folge von Moment-Anfragen für jeden Zeitpunkt *t'* nach *t*. Es wird zu jedem Zeitpunkt *t'* eine neue Moment-Anfrage gestellt. Die kontinuierliche Anfrage „finde alle Hotels im Umkreis von 3km“ wird eine Antwortmenge generieren, die sich über den Zeitverlauf dynamisch ändern kann, auch wenn die Datenbank nicht explizit aktualisiert wird. Einsetzbar ist sie z.B. als dauerhaft laufende Anfrage, die einem Autofahrer permanent alle Hotels im Umkreis anzeigt. Natürlich wäre es höchst ineffizient, die Anfrage wirklich zu jedem Zeitpunkt neu auszuwerten. Anstelle dessen wertet man die eigentliche Anfrage nur einmal aus und liefert eine Menge von Tupeln mit zugehörigen Zeitstempeln zurück. Jedes Tupel stellt eine erfüllende Instanz der Pädikatsvariablen dar. Seine zugehörigen Zeitstempel geben an, in welchem Zeitraum es der gesuchten Ergebnismenge angehört. Mit fortschreitender Zeit werden Tupel, deren Zeitperiode betreten wurde, dynamisch zur Ergebnismenge hinzugefügt, solche deren Zeit abgelaufen ist, werden entfernt. Wird eine Aktualisierung der Datenbank, wie eine Änderung des Bewegungsvektors des Autos durchgeführt, so muss die kontinuierliche Anfrage neu ausgewertet werden.

[3] In der englischen Literatur „instantaneous“ genannt.

Persistente Anfragen
Eine persistente Anfrage zum Zeitpunkt *t* ist wie eine kontinuierliche Anfrage eine Folge von Moment-Anfragen, mit dem Unterschied, dass sie alle gleichzeitig am Zeitpunkt *t* starten und danach kontinuierlich ausgewertet werden. Eine Anfrage des Typs „finde alle Fahrzeuge, die in den letzten zwei Minuten ihre Geschwindigkeit verdoppelt haben“ lässt sich nur mit einer persistenten Anfrage beantworten, da sowohl die Moment-Anfrage als auch die kontinuierliche Anfrage die zu jedem Zeitpunkt aktuelle Geschwindigkeit für alle zukünftigen Zeitpunkte annehmen. Hat ein Fahrzeug zum Zeitpunkt t_1 die Geschwindigkeit 50 und nach einer Aktualisierung zum Zeitpunkt t_2 die Geschwindigkeit 100, wird die kontinuierliche Anfrage es nicht als Ergebnis aufführen. Zum Zeitpunkt t_1 ist für alle zukünftigen Zustände die Geschwindigkeit konstant 50, also ist das Fahrzeug nicht in der Ergebnismenge. Ab Zeitpunkt t_2 ist die Geschwindigkeit für alle weiteren Zeitpunkte konstant 100, also ist das Fahrzeug weiterhin nicht in der Ergebnismenge. Die persistente Anfrage hingegen liefert das Fahrzeug als Ergebnis, da durch den Vergangenheitsbezug zum Zeitpunkt *t* die Verdoppelung der Geschwindigkeit erkannt wird. Anzumerken ist jedoch, dass im MOST-Modell eine Speicherung vergangenheitsbezogener Werte nicht vorgesehen ist, da die Werte der Subattribute bei jeder Aktualisierung überschrieben werden. Daher ist dieser Anfragetyp nicht im MOST-Modell vorhanden.

3.2 Anfragesprachen für bewegliche Objekte

Im Rahmen des DOMINO-Projekts wird eine temporale Anfragesprache namens „Future Temporal Logic“ (kurz FTL) vorgestellt, mit der sich auf einfache Art Anfragen bezüglich zukünftiger Datenbankzustände formulieren lassen. Mit ihr lassen sich Moment-Anfragen auf dem MOST-Datenmodell realisieren, jedoch werden kontinuierliche Anfragen vom Sprachumfang nicht direkt unterstützt. Diese lassen sich jedoch auf der Applikationsebene umsetzen, da sie, wie in Abschnitt 3.1 erwähnt, nur Erweiterungen von Moment-Anfragen darstellen. Die Sprache ist darauf ausgelegt, auf einem vorhandenen Datenbank-Management-System aufzubauen, dass schon eine herkömmliche nichttemporale Anfragesprache, wie SQL bereitstellt, die jedoch um räumliche Prädikatsfunktionen erweitert sein muss. Geeignet dafür sind räumliche Datenbanksysteme, da diese oft genutzte Prädikate wie Abstände, Schnittpunkte und Schnittflächen von räumlichen Objekten schon bereitstellen. In der SQL-Syntax eingebettet kommen FTL-Sprachelemente nur in der WHERE-Klausel vor, in der sie zeitliche Bedingungen ausdrücken.

Die Syntax und die Semantik von FTL sind der Prädikatenlogik erster Stufe sehr ähnlich, mit dem Unterschied, dass es keine All- und Existenzquantoren gibt, sondern nur einen Zuweisungsquantor. Eine ausführliche formale Definition von Syntax und Semantik ist in [3] zu finden. Im Rahmen dieser Ausarbeitung werden nur die Grundlagen vorgestellt, die für das allgemeine Verständnis dieses Sprachkonzepts nötig sind.

Wie in der Prädikatenlogik erster Stufe gibt es eine Menge von Symbolen, bestehend aus Konstanten, Junktoren (wie $\wedge$ und $\vee$), der Negation $\neg$, n-stelligen

Funktionen (wie die Rechenoperationen +, - usw.), n-stelligen Prädikaten (wie <, > usw.), Variablen und Klammern. Hinzu kommen der Zuweisungsquantor ← sowie zeitbezogene Operatoren wie *until* und *nexttime*. Auf der Menge der Symbole werden weitergehend induktiv Terme und Formeln definiert. Der Einfachheit wegen wird im weiteren Text nicht mehr zwischen einem Symbol und seiner Interpretation unterschieden. Für ein Konstantensymbol *c* wird also auch dessen Wert mit *c* ausgedrückt. An dieser Stelle wird nur auf die spezielle Semantik der zeitbezogenen Operatoren und des Zuweisungsoperators eingegangen, da sie das Hauptkonzept der Sprache darstellen.

Seien s_a, *s'*, s_i Datenbankzustände der Datenbankhistorie *H*, wobei s_a der aktuelle Zustand der Datenbank ist. Eine Variablenbelegung μ im Zustand *s* wird verkürzt mit (s, μ) angegeben. Des Weiteren seien *F*, *G* zwei Formeln, *x* eine Variable und *t* ein Term der FTL.

- *F until G* besagt, dass entweder *G* zum aktuellen Zustand erfüllt ist oder es einen zukünftigen Zustand gibt, an dem *G* erfüllt sein wird und, bis es soweit ist, *F* erfüllt ist. *F until G* ist genau dann erfüllt, wenn gilt:
  Entweder ist G in (s_a, μ) erfüllt oder es existiert ein zukünftiger Datenbankzustand *s'*, so dass gilt: G ist in (s', μ) erfüllt und für alle Zustände s_i, mit $s_i < s'$ ist *F* in (s_i, μ) erfüllt.

- *Nexttime F* ist genau dann erfüllt, wenn gilt:
  F ist in (s', μ) erfüllt, wobei *s'* der Zustand ist, der sofort nach dem aktuellen Zustand s_a eintritt.

- $[x \leftarrow t](F)$ ist genau dann erfüllt, wenn gilt:
  F ist in (s_a, μ) erfüllt, wobei jedes Auftreten der Variable *x* durch den Term *t* ersetzt wird

Mit der gegebenen Semantik von *until* und *nexttime* lassen sich nun auch weitere davon abgeleitete zeitbezogene Operatoren wie *eventually*, *until_within_c* oder *eventally_within_c* definieren.

- *Eventually F* bedeutet, dass *F* in einem beliebigen zukünftigen Zustand erfüllt sein wird. Wird *true* als das Boolesche „Wahr" definiert, so gilt:
  $Eventually\ F \Leftrightarrow true\ until\ F$

- *F until_within_c G* ist wie *F until G* zu verstehen, jedoch mit einer Zeitbeschränkung von *c* Zeiteinheiten. Es gibt also einen zukünftigen Zustand, der maximal *c* Zeiteinheiten in der Zukunft liegt, in dem *G* erfüllt sein wird, wobei bis dahin *F* erfüllt ist. Wenn man *Time* als den Zeitstempel des Datenbankszustands ansieht, gilt:
  $F\ until\_within\_c\ G \Leftrightarrow [t \leftarrow Time](F\ until\ (G \wedge (Time < t+c)))$

- *Eventually_within_c F* bedeutet, dass *F* innerhalb von maximal *c* Zeiteinheiten erfüllt sein wird. Es gilt:

Eventually_within_c F $\Leftrightarrow$ *true until_within_c F*

Mit diesen Sprachkonstrukten lassen sich jetzt unter Zuhilfenahme von räumlichen Funktionen und Prädikaten Moment-Anfragen bilden. Sei die Funktion *route_dist* definiert, welche die Länge des kürzesten Weges zwischen zwei Punkten in einem Routennetzwerk (z.B. einer Straßenkarte) angibt, sowie das Prädikat *inside*, das überprüft, ob ein Punkt im Routennetzwerk in einem räumlich ausgedehnten Objekt (wie einem Polygon) enthalten ist. Weiterhin sei in einer Datenbank eine Tabelle *lastwagen* enthalten, in der alle Lastwagen einer Spedition aufgelistet sind. Die Tabelle enthalte weiter den Primärschlüssel *id*, die Nummer des Lastwagens sowie das dynamische Attribut *loc*, die aktuelle Position des Lastwagens auf seiner Route und das räumliche Attribut *ziel*, den Zielort der Route, dargestellt als Punkt im Routennetzwerk. Sei außerdem das Gebiet einer Stadt durch das Polygon *S* dargestellt.

Die folgende Anfrage liefert die Nummer aller Lastwagen, die in den nächsten 20 Minuten ihren Zielort erreichen werden:

SELECT id FROM lastwagen l WHERE eventually_within_20 (dist(loc, t.ziel)=0)

Diese Anfrage liefert die Nummern und Positionen aller Lastwagen, die in den nächsten 20 Minuten im Stadtgebiet weniger als 2000 Entfernungseinheiten von ihrem Zielort entfernt sein werden:

SELECT id, loc FROM lastwagen l
WHERE eventually_within_20 (inside(l.loc, S) $\wedge$ *dist(loc, t.ziel)<2000)*

Das grundlegende Auswertungsprinzip für die Anfragen funktioniert dahingehend, dass für jede Formel *F* eine Relation $R_F(x_1, x_2, \ldots, x_{n,}, t_{start}, t_{end})$ angelegt wird, die für jede Variable der Formel ein Attribut besitzt sowie zwei weitere Attribute, welche ein zeitliches Intervall beschreiben. Ein Tupel der Relation stellt eine erfüllende Belegung der in der Formel enthaltenen Variablen dar, sowie den Zeitraum, in dem sie gültig ist. Eine ausführliche Beschreibung der Auswertung ist in [3] zu finden.

3.3 Ungenauigkeit in MOD-Systemen

Auch mit dem in Kapitel 2.1 vorgestellten Konzept des dynamischen Attributs muss ein bewegliches Objekt zu gewissen Zeitpunkten seine Position und seinen Bewegungsvektor an das MOD-System übermitteln. Dies kann geschehen, wenn es seine Richtung oder Geschwindigkeit ändert, was jedoch je nach Anwendungsgebiet immer noch recht häufig sein kann. Ein anderer Ansatz ist, nur dann eine Aktualisierung durchzuführen, wenn die in der Datenbank gespeicherte Objektposition sich zu sehr von der wirklichen unterscheidet, also die Ungenauigkeit ein gewisses Maß überschreitet. Hierbei lassen sich zwei verschiedene Konzepte

unterscheiden, die sich beide mit der Ungenauigkeit beschäftigen. Die Abweichung[4] eines beweglichen Objekts zu einem bestimmten Zeitpunkt ist der Abstand zwischen der wirklichen und der in der Datenbank gespeicherten Lokation. Die Unsicherheit[5] eines beweglichen Objekts zu einem bestimmten Zeitpunkt gibt die Größe des Bereichs an, in dem sich alle möglichen aktuellen Positionen des Objekts befinden könnten. Sendet ein Objekt beispielsweise nur dann seine neue Position über ein Funknetzwerk an das MOD-System, wenn sich seine aktuelle Position mehr als 100 Meter von der gespeicherten unterscheidet, so ist die Unsicherheit die Größe eines Kreises mit Radius 100 Meter um den Punkt der letzten Aktualisierung.

Sieht man die Abweichung und die Unsicherheit als Kosten an, mit der sich das möglicherweise Treffen von falschen Entscheidungen bewerten lässt, kann man ein Kostenmodell aufbauen, wobei als dritter Kostenfaktor noch die Belastung der Datenbank und des Funknetzwerks durch Aktualisierungen herangezogen wird. Wenn ein Objekt immer dann eine Aktualisierung an das MOD-System schicken soll, wenn die Unsicherheit der Position einen bestimmten in *loc.uncertainity* abgespeicherten Wert *k* übersteigt, lässt sich mit Hilfe des Kostenmodells ein von der Geschwindigkeit des Objekts abhängiger Wert für *k* finden, der mit minimalen Kosten verbunden ist. Wird *k* bei jeder Aktualisierung abhängig von der Geschwindigkeit neu berechnet (die so genannte „adaptive dead-reckoning update policy“), so können die Kosten gegenüber einem fest gewählten *k* („speed dead-reckoning update policy“) auf bis zu ein Sechstel reduziert werden [3]. Wird ein mit der Zeit abnehmender Wert für *k* gewählt („disconnection detection dead-reckoning update policy“), kann unterschieden werden, ob ein bewegliches Objekt keine Aktualisierungen mehr liefert, weil es keine Verbindung zum MOD-System hat oder ob die Abweichungsobergrenze einfach noch nicht überschritten wurde.

Bei dem im MOST-Modell verwendeten Attribut *loc* wurde die Route eines beweglichen Objekts herangezogen um seine Position zu bestimmen. Allgemein lässt sich die Bewegung eines Objekts durch einen so genannten Bewegungsablauf[6] beschreiben, welcher die zeitliche Position des Objekts im Raum angibt. Er kann sowohl die bereits zurückgelegte, als auch die (vermutete) zukünftige Strecke des Objekts beschreiben. Wird der Bewegungsablauf eines beweglichen Objekts modelliert, wird er meist als Line im dreidimensionalen Raum dargstellt, wobei zwei Dimensionen den (zweidimensionalen) Raum darstellen und die dritte für die Zeit herangezogen wird. Die Linie ist durch eine Sequenz von Punkten $<(x_1, y_1, t_1), \ldots, (x_n, y_n, t_n)>$ mit $t_1<\ldots<t_n$ repräsentiert, was in Abbildung 1 dargestellt ist. Es wird angenommen, dass ein Objekt, welches sich zum Zeitpunkt t_i an der Position (x_i, y_i) befindet, sich in der Zeitperiode $[t_i, t_{i+1}]$ auf einer geraden Linie von (x_i, y_i) nach (x_{i+1}, y_{i+1}) bewegt, wodurch die Bewegung als implizite Funktion der Zeit angesehen werden kann. Die räumliche Projektion eines Bewegungsablaufs in die xy-Ebene stellt dabei die Route des Objekts dar. Wird in dieses Modell noch für jeden Zeitpunkt t_i die Unsicherheit als horizontaler Kreis mit Radius *th* um (x_i, y_i, t_i) hinzugefügt, so

[4] In englischsprachiger Literatur „deviation“ genannt.
[5] In der englischen Literatur als „uncertanity“ zu finden.
[6] In der englischen Literatur „trajectory“ genannt.

bildet sie im Verlauf der Zeit zwischen zwei Punkten (x_i, y_i, t_i) und $(x_{i+1}, y_{i+1}, t_{i+1})$ einen dreidimensionalen Zylinder um die Linie.

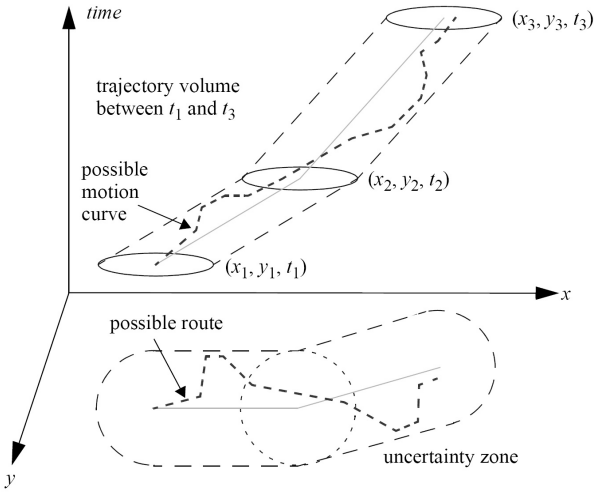

Abb. 1. Darstellung des Bewegungsablaufs eines beweglichen Objekts

Wird dieser Zylinder in die xy-Ebene projiziert, so entsteht eine Fläche, die so genannte Unsicherheitszone. Es ist zu erkennen, dass die wirkliche Route eines Objekts innerhalb der Unsicherheitszone sich stark von der geraden Linie zwischen zwei Punkten (x_i, y_i), (x_{i+1}, y_{i+1}) unterscheiden kann. Durch die Unsicherheit lassen sich bei einer Anfrage an die Position eines Objekts nun verschiedene Fälle unterscheiden. Man kann nach Objekten fragen, die sich zu einem bestimmten Zeitpunkt sicherlich oder nur vielleicht in einer bestimmten Region aufhalten oder nach Objekten die sich in einem bestimmten Zeitraum manchmal oder immer in einer Region aufhalten. Aus den Kombinationen dieser Fälle lassen sich sechs verschiedene „spatio-temporale“ Prädikate unterscheiden, die sich in Anfragesprachen integrieren lassen. Eine genaue Beschreibung dazu liefert [3].

4 Indexstrukturen

Möchte man die im letzten Kapitel aufgezeigten Anfragen effizient durchführen, so benötigt man Indexstrukturen. Herkömmliche Indextechnologien für räumliche Datenbanken, wie beispielsweise der R-Baum, sind jedoch auf statische räumliche Objekte ausgelegt und eignen sich nicht für bewegliche Objekte. Mittlerweile gibt es mehrere Arbeiten, die sich mit der Indexierung beweglicher Objekte beschäftigen, wobei [11] einen Überblick gibt. Viele der Ansätze basieren auf Baumstrukturen, wie beispielsweise vom R-Baum abgeleitete Datenstrukturen wie der Time-Parametrized R-Tree (kurz TPR-Tree). Jedoch gibt es in der Literatur kaum Strukturen, die sich sowohl zur Indexierung von vergangenen und aktuellen als auch von zukünftigen

Objektpositionen gleichzeitig benutzen lassen. Ein dies erfüllender Vorschlag ist die auf dem B^x-Baum basierende „Broad B^x Index Technology“ (kurz BB^x-Index), welche ähnlich wie das MOST-Modell Bewegungsvektoren zur Darstellung der Objektlokation verwendet. In Abschnitt 4.1 werden die Grundlagen des B^x-Baums erklärt und in 4.2 der auf ihm basierende BB^x-Index erläutert. In Abschnitt 4.3 werden einige Optimierungen für B-Bäume vorgestellt, damit diese mit hohen Aktualisierungsraten fertig werden können. Letztendlich wird in Abschnitt 4.4 mit dem „Lazy-Update-Grid“ ein nicht-baumbasierter Lösungsansatz vorgestellt, welcher jedoch nur die aktuelle Position eines Objekts verwalten kann.

4.1 Der B^x-Baum

Der von Lin et al. [6] vorgestellte B^x-Baum ermöglicht es, die aktuellen und zukünftigen Positionen von Objekten zu indexieren. Er baut auf dem klassischen B^+-Baum auf, der in vielen heutigen Datenbanksystemen zum Einsatz kommt und sich durch einen hohen Verzweigungsgrad, vergleichsweise gute Skalierbarkeit und Effizienz im Bezug auf Anfragen und Aktualisierungen bewährt hat. Die Daten, in diesem Fall die Lokationen der beweglichen Objekte, werden ausschließlich in den Blättern abgelegt.

Da ein B^+-Baum eine eindimensionale Indexstruktur ist, müssen die mehrdimensionalen räumlichen Positionsangaben auf eindimensionale Werte abgebildet werden, um sie im Baum speichern zu können. Dies wird durch Anwendung einer so genannten raumfüllenden Kurve bewerkstelligt. Eine raumfüllende Kurve, auch FASS-Kurve[7] genannt, ordnet jedem Punkt in einem diskreten mehrdimensionalen Raum einen eindeutigen Wert zu [14]. Die Kurve durchläuft dabei jeden Punkt des Raums durch Wiederholung ihres Konstruktionsverfahrens exakt einmal, ohne sich selbst zu überschneiden. Beispiele für FASS-Kurven im zweidimensionalen Raum sind die Peano-Kurve sowie die Hilbert-Kurve, welche beide in Abbildung 2 dargestellt sind.

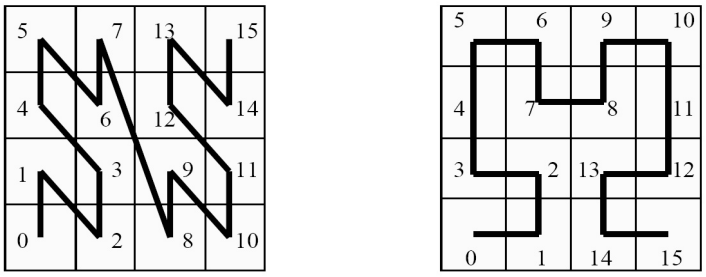

Abb. 2. Schematische Darstellungen einer Peano-Kurve (links) und einer Hilbert-Kurve (rechts) im zweidimensionalen Raum

[7] Vom englischen "space-**f**illing, self-**a**voiding, **s**imple und self-**s**imilar".

Beide haben die Eigenschaft, dass sie zu einem gewissen Maß zwei in der Ebene nah beieinander liegende Punkte wieder auf zwei nah gelegene Werte abbilden, also die räumliche Nähe erhalten. Das „x“ im Namen B^x-Baum soll symbolisieren, dass jede beliebige raumfüllende Kurve „x“ in dem Verfahren eingesetzt werden kann, wobei in [6] darauf verwiesen wird, dass mit der Hilbert-Kurve im Vergleich zur Peano-Kurve im zweidimensionalen Raum eine bessere Performanz erreichbar ist.

Wendet man eine FASS-Kurve auf die Position eines beweglichen Objekts an, so hat man jedoch durch die konstante Bewegung wieder das Problem der hohen benötigten Aktualisierungsraten. Deshalb wird, ähnlich wie beim MOST-Datenmodell, der Bewegungsvektor dazu herangezogen, um die Lage zu bestimmen. Es wird eine lineare Funktion der Zeit benutzt, um die Objektpositionen dynamisch anzugeben. Die Lokation wird dabei als (x, v, t_u) dargestellt, wobei x die Ausgangsposition zum Aktualisierungszeitpunkt t_u und v der Bewegungsvektor ist. Durch diese Darstellung der Positionen wird die benötigte Aktualisierungshäufigkeit für Objekte wie Flugzeuge, welche oft für vergleichsweise lange Zeiträume einen konstanten Bewegungsvektor haben, deutlich reduziert [6]. Für Objekte, welche dauernd ihre Geschwindigkeit und Bewegungsrichtung ändern, ergeben sich jedoch keine Verbesserungen.

Im B^x-Baum wird weitergehend der Index in Partitionen bezüglich der Aktualisierungszeit unterteilt. Dabei wird die Zeitachse in Intervalle zerlegt, deren Länge der maximalen Zeit Δt_{mu} zwischen zwei Aktualisierungen einer Objektlokation entspricht. Dies ist die Zeit, welche maximal vergehen darf, bis ein Objekt seinen aktuellen Ort neu an das DBMS übermitteln muss, also ein zeitliches Maß der Unsicherheit. Jede so eingeteilte Partition hat dabei eine eindeutige Partitionsnummer. Diese Intervalle werden dann weiter in n gleich lange Subintervalle gegliedert, so genannte Phasen, wobei n durch die minimale Zeitspanne t_{min} bestimmt wird, in der ein Objekt seine Lage aktualisieren kann. Für $n = 2$ lässt sich in der Praxis nach [6] ein besseres zeitliches Verhalten erzielen als für andere Werte, weswegen dieser Wert bei weiteren Betrachtungen gewählt wird. Jeder Phase ist ein Zeitstempel t_{lab} zugeordnet, der den Zeitpunkt angibt, an dem die Phase endet. Allen Aktualisierungszeitpunkten einer Phase wird dabei ein gleicher Zeitstempel zugeordnet.

Der in einem Blattknoten des B^x-Baum abgespeicherte Indexwert setzt sich wie folgt zusammen: Er ist die Konkatenation der binären Repräsentationen der Partitionsnummer *partition* mit dem Wert *xrep* der raumfüllenden Kurve für die Position des Objekts zum durch die Phase bestimmten Zeitpunkt t_{lab}.

$$B^x value = partition_2 \oplus xrep_2$$

Die Werte werden im Detail wie folgt berechnet:

Wenn ein Objekt eine Aktualisierung der Position zum Zeitpunkt t_u an das MOD-System übermittelt, so wird aus t_u die Phase bestimmt, in der das Objekt indexiert und eingefügt werden soll, sowie das zur Phase gehörende t_{lab} berechnet. Dabei ist zu

beachten, dass die Objektpositionen zum Zeitpunkt t_{lab} indexiert werden, welcher etwas später als ihr eigentlicher Änderungszeitpunkt ist. In der folgenden Formel liefert $\lceil x \rceil$ den jeweils nächsten zukünftigen Phasen-Zeitstempel zurück:

$$t_{lab} = \lceil t_u + \Delta t_{mu} / n \rceil$$

Dieser Mechanismus wird in Abbildung 3 für $n = 2$ dargestellt. Die Phasen haben hierbei eine entsprechende Phasenlänge von $0{,}5\Delta t_{mu}$. Allen Objekte mit Aktualisierungszeitpunkt $0 < t_u < 0{,}5t_{mu}$ wird der Zeitstempel $t_{lab} = t_{mu}$ zugeordnet, solchen mit Aktualisierungszeitpunkt $0{,}5t_{mu} < t_u < t_{mu}$ wird $t_{lab} = 1{,}5t_{mu}$ zugeordnet usw.

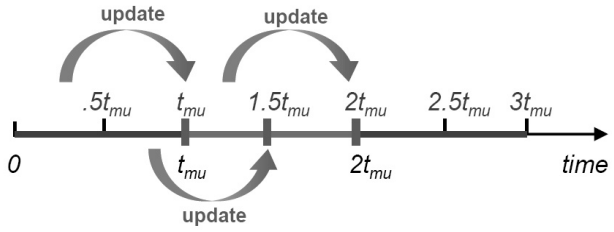

Abb. 3. Phasen im B^x-Baum

Aus t_{lab} lässt sich nun die übergeordnete Partitionsnummer berechnen:

$$partition = (t_{lab} / (\Delta t_{mu} / n) - 1) \bmod (n+1)$$

Für das Objekt wird nun die zukünftige Lokation zum Zeitpunkt t_{lab} bestimmt, die es bei gleichbleibender Bewegung haben würde. Der Wert berechnet sich dabei aus der Position $\vec{x}$ und dem Bewegungsvektor $\vec{v}$ zum Aktualisierungszeitpunkt t_u und der Zeitdifferenz zwischen t_u und t_{lab}. Diese mehrdimensionale Positionsangabe wird dann mit Hilfe der raumfüllenden Kurve auf einen eindimensionalen Wert *xrep* abgebildet:

$$xrep = FASS(\vec{x} + \vec{v}(t_{lab} - t_u))$$

Die Binärrepräsentationen der Partitionsnummer und des Positionswerts der FASS-Kurve werden nun konkateniert und ergeben den Indexwert:

$$B^x value = partition_2 \oplus xrep_2$$

Die indexierten Positionen in einer Phase beziehen sich demnach alle auf denselben Zeitpunkt t_{lab}, wodurch die durch die FASS-Kurve erhaltene räumliche Nähe weiterhin bestehen bleibt. Die Darstellung 4 gibt einen solchen Einfügevorgang für $n = 2$ und einer Hilbert-Kurve in der Ebene beispielhaft wieder.

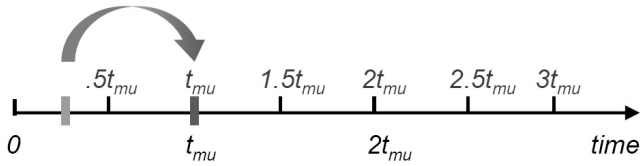

Abb. 4(a). Einfügen eines Objekts in den B^x-Baum

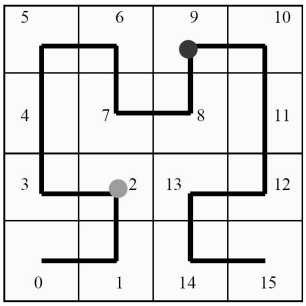

Abb. 4(b). Bestimmung von Objektpositionen mit Hilfe der Hilbert-Kurve

In Abbildung 4(a) aktualisiert ein Objekt seine Lage zum hell hervorgehobenen Zeitpunkt t_u = 0,25. Die Aktualisierung bekommt dann den Zeitstempel t_{lab} = 1 (dunkel hervorgehoben) zugeordnet, woraus sich der die Partitionsnummer *partition = 1* berechnen lässt, welcher der Binärwert 1_2 entspricht. In Abbildung 4(b) hat das Objekt zum Zeitpunkt t_u die hell hervorgehobene räumliche Position. Die dunkel hervorgehobene zukünftige Lage zum Zeitpunkt t_{lab} erhält man über die lineare Bewegungsfunktion des Objekts. Die Hilbert-Kurve liefert für diese räumliche Position den Wert *xrep* = 9, mit Binärrepräsentation 1001_2. Durch Konkatenation der beiden binären Werte 1_2 und 1001_2 erhält man den Indexwert 11001_2, welcher 25 entspricht.

Da die Lokationsänderungen immer zum Endzeitpunkt t_{lab} einer Phase gespeichert werden, stellt sich die Frage, wie Anfragen zu beliebigen Zeitpunkten verarbeitet werden können, da die Positionen zu den Anfragezeiten in den meisten Fällen nicht mit denen der Phasenendzeiten übereinstimmen werden. Dazu werden bei räumlichen Anfragen die Anfragefenster so erweitert, dass sie Objekte einschließen können, welche die Bedingung zum Zeitpunkt t_{lab} nicht, sie aber zum Anfragezeitpunkt vielleicht erfüllen. Da im B^x-Baum die Lokationen zu einem späteren Zeitpunkt als der Aktualisierungszeit indexiert werden, müssen bei einer Erweiterung des Fensters die Objektpositionen entweder zu einem vorherigen oder zu einem späteren Zeitpunkt betrachtet werden.

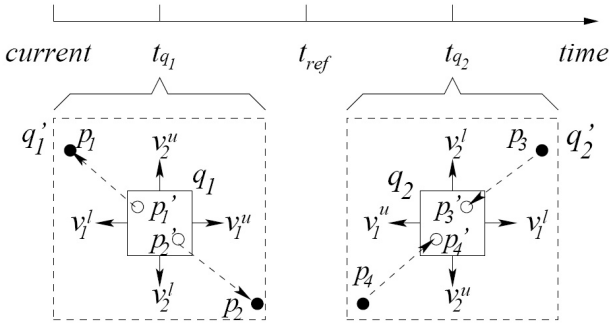

Abb. 5. Erweiterung des Anfragefensters im B^x-Baum

Die beiden Arten der Vergrößerung sind in Abbildung 5 verdeutlicht. Die beweglichen Objekte p_1 bis p_4 sind im B^x-Baum indexiert. q_1 und q_2 sind zwei Anfragen zu den Zeitpunkten t_{q1} und t_{q2}. In der Grafik wird durch die unausgefüllen Kreise p_1' bis p_4' angezeigt, dass die Objekte zu den Anfragezeitpunkten in den Ergebnismengen q_1 und q_2 (durchgezogene Rechtecke) liegen sollten, aber nur durch eine Ausweitung der Anfragefenster q_1' und q_2' (gestrichelte Rechtecke) erreicht werden können. Eine detaillierte Beschreibung der Bestimmung der Größe der Anfragefenster findet sich in [6].

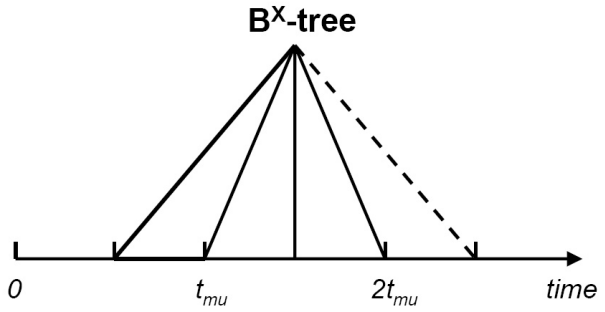

Abb. 6. Schematische Darstellung eines B^x-Baums

Im B^x-Baum existieren zu einem Zeitpunkt maximal n+1 Phasen gleichzeitig [6], wobei n die Anzahl der Phasen pro Partition ist. Mit voranschreitender Zeit wird jeweils die älteste Phase aus dem Baum entfernt und eine neue Phase hinzugefügt. Dies soll durch die gestrichelte Linie in Darstellung 6 verdeutlicht werden. Dieser Mechanismus wird laut [5] eingesetzt, um eine effiziente Behandlung der Zeit zu ermöglichen. Aktualisiert ein Objekt nicht innerhalb von Δt_{mu} seine Position, so wird es von der ältesten in die neueste Partition migriert. Daher ist mit dem B^x-Baum nur eine Betrachtung aktueller und zukünftiger Positionen möglich.

4.2 Der BB^x-Index

Aufbauend auf dem B^x-Baum wurde von Lin et al. [5] die „Broad B^x Index Technology“ (kurz BB^x-Index) vorgestellt, welche vergangene, aktuelle und zukünftige Objektpostionen verwalten kann. Es wurden dabei viele Konzepte des B^x-Baums übernommen, wie die Partitionierung der Zeitachse in Phasen und die Repräsentation der aktuellen Objektlagen durch Verwendung einer linearen Bewegungsfunktion und einer raumfüllenden Kurve. Da die Berechnungen meist identisch mit denen aus 3.1 sind, werden sie hier nicht noch einmal aufgeführt. Jedoch gibt es den Unterschied, dass der BB^x-Index aus einem Wald von Bäumen besteht und die Adressierung einzelner Werte etwas anders gehandhabt wird. Um im Gegensatz zum B^x-Baum auch historische Objektpositionen verwalten zu können, repräsentiert jeder einzelne Baum des Walds eine zeitliche Phase.

Die Knoten in den einzelnen Bäumen enthalten Einträge der Form (*xrep*, t_{start}, t_{end}, *pointer*). Bei Blattknoten ist *pointer* ein Zeiger auf das entsprechende bewegliche Objekt, dessen Position durch *xrep* angegeben ist, wobei *xrep* der von der raumfüllenden Kurve zurück gelieferte Positionswert ist. t_{start} gibt den Einfügezeitpunkt in den Baum an, t_{end} gibt an, zu welchem Zeitpunkt der Standort des Objekts das letzte Mal aktualisiert wurde. Bei inneren Knoten zeigt *pointer* auf einen Kindknoten der nächsten Ebene und t_{start} und t_{end} geben das minimale t_{start} und das maximale t_{end} aller Knoten des unteren Teilbaums an. Jeder Baum im BB^x-Index verfügt über einen eindeutigen Zeitstempel t_{ts} und einen Zeitraum *lifespan*, in dem er gültig ist. Der Zeitstempel gleicht dem t_{lab} des B^x-Baums und wird ebenso durch eine Partionierung der Zeitachse in Phasen erhalten. *lifespan* wird durch t_{start} der Knoten des Baums bestimmt. Er ist die Zeitspanne zwischen dem kleinsten t_{start} und dem größten t_{start} plus der maximalen Zeit zwischen zwei Aktualisierungen Δt_{mu}. Die Wurzeln der jeweiligen Bäume sind in einem Array abgespeichert und werden über *lifespan* adressiert.

Alle Objekte, die zeitlich in einer Phase eingefügt wurden, bekommen das gleiche t_{ts} zugeordnet, das dem Endzeitpunkt ihrer Phase entspricht. Genauer gesagt, bekommt ein Objekt, welches zum Zeitpunkt t_{start} seine Positionsangaben aktualisiert hat, das kleinste t_{ts} zugeordnet, das größer als t_{start} ist. Auch werden eingefügte Objekte nach den ihnen zugeordneten t_{ts} indexiert. Dabei wird wieder nicht ihr Standort zum Zeitpunkt t_{start} in den Index aufgenommen, sondern der zum Zeitpunkt t_{ts}, welcher mit Hilfe der linearen Bewegungsfunktion berechnet wird. Der mehrdimensionale Positionswert wird danach durch Anwendung der raumfüllenden Kurve in einen eindimensionalen Wert überführt, der dann in den Index aufgenommen wird. Im Gegensatz zum B^x-Baum wird jedoch keine Konkatenation der zeitlichen und räumlichen Werte durchgeführt, sondern die zeitliche Komponente t_{ts} bestimmt den Baum, in den der räumliche Wert eingefügt wird.

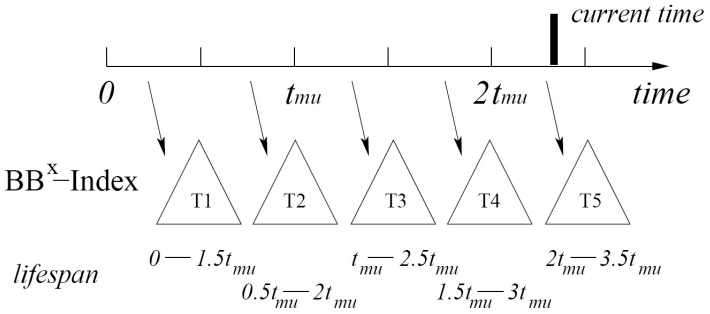

Abb. 7. Schematische Darstellung des BB^x-Index

In Abbildung 7 ist beispielhaft zu sehen, wie Objekte in die einzelnen Bäume eingefügt werden, wobei wieder $n = 2$ gilt. Objekte, die zwischen den Zeitstempeln 0 und $0{,}5t_{mu}$ eingefügt wurden, werden mit ihrer Lokation zum Zeitpunkt $0{,}5t_{mu}$ in Baum T_1 indexiert. Solche, die zwischen $0{,}5t_{mu}$ und t_{mu} lagen, werden mit dem Standort, den sie zum Zeitpunkt t_{mu} haben, in Baum T_1 abgelegt usw. Der Baum T_1 hat eine maximale Gültigkeitsdauer (*lifespan*) von 0 bis $1{,}5t_{mu}$, da die Objekte mit dem kleinsten Zeitstempel zum Zeitpunkt 0 eingefügt wurden und die neuesten zum Zeitpunkt $0{,}5t_{mu}$. Letztere können maximal eine Zeitspanne von Δt_{mu} bestehen bleiben, bis eine neue Positionsaktualisierung durchgeführt wird, was spätestens am Zeitpunkt $1{,}5t_{mu}$ geschieht.

Mit dem BB^x-Index können nun Anfragen auf vergangene, aktuelle und zukünftige Objektpostionen durchgeführt werden, wobei, wie auch beim B^x-Baum der Ausblick in die Zukunft durch Δt_{mu} beschränkt ist. Mit einem vergrößerten Δt_{mu} kann man weiter in der Zukunft liegende Objektlagen voraussagen, jedoch erkauft man sich dies durch eine höhere Unsicherheit der Ergebnisse.

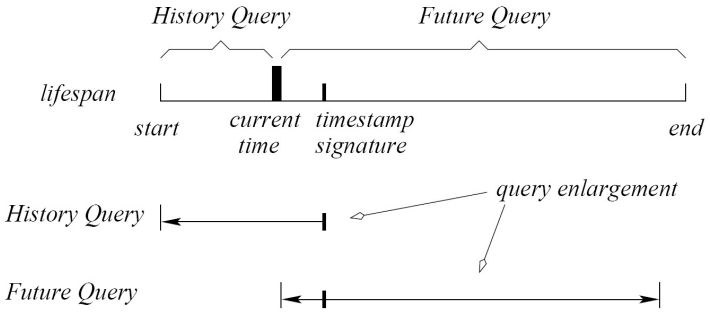

Abb. 8. Mögliche Anfragen im neuesten Baum des BB^x-Index

Im neuesten Baum des BB^x-Index mögliche Anfragen sind in Abbildung 8 dargestellt. Vergangenheitsbezogene Anfragen beginnen beim Einfügezeitpunkt *start* des ältesten Objekts im Baum (wohl gemerkt, dies ist der aktuelle Baum, es gibt davor ggf. noch eine große Anzahl weiter in der Vergangenheit liegender Bäume) und

gehen bis zum aktuellen Zeitpunkt. Zukunftsbezogene Anfragen reichen bis maximal Δt_{mu} in die Zukunft. Wie im B^x-Baum muss auch hier eine Erweiterung des Anfragefensters durchgeführt werden, um alle möglicherweise die Anfragebedingung erfüllenden Objekte zu finden.

4.3 Optimierungen in B-Bäumen zur Handhabung hoher Aktualisierungsraten

Beim B^x-Baum und dem BB^x-Index wird versucht, durch Verwendung des Bewegungsvektors die Anzahl der Aktualisierungen zu minimieren. Jedoch können auch hier immer noch hohe Aktualisierungsraten anfallen, was verbunden mit der Transkationsverwaltung und dem „Logging“ zu hohen Belastungen des Datenbanksystems führt. Da in MOD-Systemen der „Location-Management-Perspektive“ Aktualisierungen wesentlich häufiger als Anfragen vorkommen, sollten Optimierungen speziell am Aktualisierungsverhalten der Indexstrukturen ansetzen. In [2] wird ein Überblick über Ansätze zur Handhabung hoher Aktualisierungsraten in B-Baumstrukturen gegeben, die alle gemeinsam haben, dass sie meist durch Inkaufnahme einer verminderten Anfrageperformanz bessere Ergebnisse bei Aktualisierungen erreichen. In diesem Abschnitt werden nun die Grundlagen dieser Konzepte kurz aufgelistet.

Eine erste Gruppe von Optimierungen setzt bei Verbesserungen der E/A-Geschwindigkeit an. Ein generischer Ansatz, der sich nicht nur bei B-Bäumen, sondern sich auch allgemein einsetzen lässt, ist es, die Daten und zusätzlich auch das Transaktions-Log zu komprimieren. Weiterhin können E/A-Operationen asynchron durchgeführt werden, wie beispielsweise durch „Write-Behind-Caching“. Hierbei werden veränderte Cache-Einträge um ein gewisses Zeitintervall verzögert und asynchron in den Permanentspeicher geschrieben. Dieses Verfahren erhöht zwar nicht den Durchsatz des Datenbanksystems, ermöglicht aber oft große Schreiboperationen, die effizienter als viele kleine sind. Außerdem können so Spitzen in der Arbeitsbelastung abgefangen werden. Bei schreiboptimierten B-Bäumen gibt es eine dynamische Platzierung der Daten im Permanentspeicher. Hierbei werden die Seiten so gespeichert, dass verschiedene Schreiboperationen alle im selben Bereich der Festplatte ausgeführt werden, wodurch mehrere kleine Schreiboperationen in einer großen sequentiellen Aktion ausgeführt werden können.

Eine zweite Gruppe von Optimierungen setzt bei der Pufferung von Einfügeoperationen im Baum an. Dies kann entweder durch Pufferung von Knoten im Baum selbst oder in separaten Datenstrukturen, wie zusätzlichen B-Bäumen oder Hash-Tabellen geschehen. Durch Partitionierung des B-Baums mit einer zusätzlichen führenden Stelle im Schlüssel kann auch eine Pufferung durchgeführt werden. Dabei werden einige der Partitionen als Puffer für den restlichen Baum benutzt, so dass neu in den Baum eingefügte Objekte im Hauptspeicher gehalten werden können. Änderungen oder Löschungen von bestehenden Knoten können mit Hilfe eines so genannten „Differential File“ optimiert werden. Hierbei werden geänderte Eintrage in den Baum hinzugefügt, ohne dass die veralteten Einträge entfernt oder modifiziert

werden. Das „Differential File“ gibt hierbei an, welche Version eines Eintrags die aktuelle ist.

Letztendlich könnte man auch die transaktionalen Zusicherungen des Datenbanksystems für die Indexstrukturen lockern, um eine höhere Performanz des Systems bei reduzierter Sicherheit zu erreichen. Es ließe sich beispielweise die Anforderung der Atomarität abschwächen, was ermöglichen würde, mehrere Aktualisierungen bzw. Transaktionen in einer großen Transaktion auszuführen. Ein anderer Ansatz wäre es, die Zusicherungen für den Fehlerfall abzuschwächen und auf „Logging“ der Indexstrukturen in bestimmten Fällen zu verzichten. Diese Ansätze bergen aber die große Gefahr, dass Daten inkonsistent werden können, und es bleibt abzuwägen, ob sich ihr Einsatz in bestimmten Anwendungsszenarien daher wirklich lohnt.

4.4 Das LUGrid

Die vorgestellten Verfahren B^x-Baum und BB^x-Index benutzen beide Baumstrukturen und versuchen die Anzahl der Aktualisierungen zu minimieren. Bei dem von Xiong et al. [15] vorgestellten „Lazy-Update Grid-based Index“ (kurz LUGrid) wird eine gänzlich andere Herangehensweise an die Problematik gewählt. Objekte übermitteln weiterhin ihre absolute Position an das MOD-System, wodurch eine hohe Aktualisierungsrate in Kauf genommen wird, jedoch durch Einsatz der „Lazy-Update-Technik“ mit stark verminderten Zugriffskosten im Datenbanksystem. Das LUGrid, welches nur für die Indexierung aktueller Lokationen verwendet werden kann, basiert auf einer multidimensionalen Indexstruktur namens „Grid File“ [13]. Die Hauptkonzepte der „Lazy-Update-Technik“ sind die beiden Verfahren „Lazy-Insertion“ und „Lazy-Deletion“, welche vergleichbar mit dem in Abschnitt 3.3 erwähnten „Differential File“ sind. Bei „Lazy-Insertion“ werden die benötigten Zugriffsoperationen einer Aktualisierung reduziert, indem eine weitere hauptspeicherbasierte Schicht über die eigentlichen Indexstrukturen gelegt wird, bei der an die gleiche Seite in der Datenbank gerichtete Aktualisierungen zusammengefasst und auf einmal ausgeführt werden. Dadurch werden die hohen Lese- und Schreibkosten vieler unabhängiger Einzelaktualisierungen vermieden, was letztendlich die umgerechneten Kosten einer Aktualisierung drastisch senkt. Durch „Lazy-Deletion“ müssen veraltete Einträge nicht erst gelöscht werden, bevor aktualisierte Werte eingefügt werden können, sondern sie existieren simultan. Das Löschen wird so lange hinausgezögert, bis die betreffende Seite mit den alten Werten durch eine Aktualisierung in den Hauptspeicher geladen wird. Dadurch werden die Kosten für das Suchen und Löschen veralteter Werte eingespart. Das LUGrid besitzt drei grundlegende Datenstrukturen: das im Hauptspeicher liegende „Memory-Grid“, das sich auf dem Permanentspeicher befindende „Disk-Grid“ und eine hauptspeicherbasierte Hash-Struktur namens „Miss-Deletion Memo“.

Das „Disk-Grid“ (kurz DG) besteht aus einer Menge sich nicht überlappender so genannter DG-Zellen, die jeweils in einer Festplatten-Seite gespeichert sind. Eine solche Zelle speichert die Positionsinformationen aller Objekte, die innerhalb ihrer

räumlichen Grenzen liegen. Eine solche Zelle lässt sich als Eintrag $(N_E, E_1, \ldots, E_n)$ mit $n > 0$ darstellen. N_E ist die Anzahl der in der Zelle gespeicherten Objekte E_1 bis E_n, wobei ein einzelnes Objekt die Form *(OID, OLoc)* besitzt. *OID* wird zur eindeutigen Identifikation des Objekts benutzt und *OLoc* stellt die letzte Position des Objekts dar, die in den Permanentspeicher geschrieben wurde.

Das „Memory-Grid“ (kurz MG) ist ein zweidimensionales Array, bei der jedes Element eine so genannte MG-Zelle ist. Jede Zelle verweist auf genau eine DG-Zelle, in der ihre ausgeschriebenen Daten gespeichert werden. Eine MG-Zelle kann dabei als hauptspeicherbasierter Puffer für die entsprechende DG-Zelle angesehen werden und eine bestimmte Anzahl an Objekten zwischenspeichern. Einer DG-Zelle können zwar mehrere MG-Zellen zugeordnet werden, welche ein Rechteck formen müssen, einer einzelnen MG-Zelle ist jedoch immer nur genau eine DG-Zelle zugeordnet. Ein Eintrag einer MG-Zelle hat die Form $(N_U, M_{Region}, D_{id}, N_E, D_{Region}, E_1, \ldots, E_m)$ mit $m > 0$. Dabei ist N_U die Anzahl der zwischengespeicherten Aktualisierungen, M_{Region} die von der MG-Zelle abgedeckte Region, D_{id} die Nummer der entsprechenden DG-Zelle, N_E die Anzahl der in der Zelle gespeicherten Objekte E_1 bis E_m und D_{Region} die von der DG-Zelle abgedeckte Region. Die Form eines einzelnen Objekts ist identisch mit der in der DG-Zelle verwendeten Form mit dem Unterschied, dass *OLoc* die letzte vom beweglichen Objekt erhaltene Position ist. Im MG werden die Objekte per doppeltem Hashing indexiert, einmal nach *OID* und einmal nach der Position.

Da im LUGrid veraltete und aktuelle Objektpositionen parallel existieren können, wird das im Hauptspeicher liegende „Miss-Deletion-Memo“ (kurz MDM) herangezogen, um die Werte zu unterscheiden. In der hash-basierten MDM-Tabelle werden alle Objekte aufgelistet, die mindestens eine Aktualisierung verpasst haben. Ein Eintrag hat die Form *(OID, OLoc, MDnum)*, wobei *OLoc* die aktuellste in das DG geschriebene Position des Objekts und *MDnum* die Anzahl der Löschungen ist, die das Objekt verpasst hat oder anders ausgedrückt, die Anzahl der in DG vorhandenen veralteten Versionen des Objekts. Wird bei einem Eintrag *MDnum* = 0, so kann er aus dem MDM entfernt werden, da nur noch die aktuelle Version des Objekts im DG vorhanden ist.

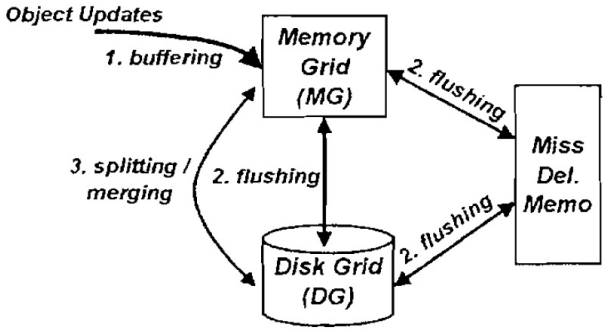

Abb. 9. Funktionsabläufe im LUGrid

In Abbildung 9 wird schematisch verdeutlicht, wie Aktualisierungen behandelt werden. Eintreffende Aktualisierungen werden im MG zwischengespeichert und so gepuffert (1). Da eine neue Aktualisierung auftreten kann, bevor der alte Wert in das DG geschrieben wurde, muss bei jedem eintreffenden Wert überprüft werden, ob ein Eintrag mit derselben *OID* schon vorhanden ist. Ist dies der Fall, so wird der alte Eintrag aus dem MG gelöscht. Der aktuelle Wert wird dann der MG-Zelle zugeordnet, deren abgedeckte Region die aktuelle Objektposition enthält. In die entsprechende DG-Zelle werden die Daten nur dann ausgeschrieben, wenn die MG-Zelle voll oder der gesamte zur Zwischenspeicherung bereitstehende Speicher belegt ist. In diesem Fall wird die am meisten gefüllte MG-Zelle in die ihr zugeordnete DG-Zelle ausgeschrieben. Das Ausschreiben eines Wertes benötigt dabei Koordination zwischen allen drei Datenstrukturen (2). Dabei wird die entsprechende DG-Zelle in den Hauptspeicher gelesen und mit Hilfe eines Vergleichs von *OID* im MDM überprüft, ob in ihr veraltete Einträge enthalten sind, deren aktuelle Positionen in anderen Zellen zu finden sind. Wird ein solcher Eintrag gefunden, so löscht man ihn aus dem DG und dekrementiert das entsprechende *MDnum* im MDM. Nachdem die veralteten Einträge entfernt wurden, wird für jeden aktualisierten Eintrag im MG der entsprechende Eintrag im DG gesucht. Wird dieser gefunden, so wird er aktualisiert und der Eintrag aus dem MG entfernt. Wird kein passender Eintrag gefunden, so muss dieser in einem anderen DG liegen und ist dort veraltet. In diesem Fall wird, falls ein MDM-Eintrag vorhanden ist, dieser mit der neuen Objektposition aktualisiert und *MDnum* inkrementiert. Ist kein passender MDM-Eintrag vorhanden, so wird ein neuer angelegt. Können auf diese Art alle neuen Einträge in die DG-Zelle eingefügt werden, so werden sie aus der MG-Zelle entfernt. Falls die DG-Zelle jedoch beim Einfügen ihre Kapazitätsobergrenze erreicht, so kann sie geteilt oder mit anderen Zellen verschmolzen werden (3). Auf diesen Mechanismus wird in dieser Ausarbeitung jedoch nicht weiter eingegangen, eine genaue Beschreibung findet sich in [15].

Der Aktualisierungsmechanismus soll nun an einem Beispiel mit Hilfe von Abbildung 10 verdeutlicht werden. Abbildung 10.1(a) zeigt ein DG mit vier Zellen *A*, *B*, *C* und *D*, in denen neun Objekte o_1 bis o_9 abgespeichert sind. Darstellung 10.1(b) zeigt das zugehörige MG an, das in sechs Zellen *1* bis *6* unterteilt ist. Den MG-Zellen *1* und *2* ist dieselbe DG-Zelle *A* zugeordnet, den MG-Zellen *3* und *6* die DG-Zelle *B*. Weiterhin gibt es momentan keine veralteten Einträge, weswegen das MDM in 10.1(c) leer ist. Eine MG-Zelle wird als voll angesehen, wenn sie zwei Einträge enthält, was bei Zelle *2* der Fall ist. Ihr Inhalt soll jetzt in die DG-Zelle *A* geschrieben werden, wobei im MDM überprüft wird, ob in *A* veraltete Einträge enthalten sind. Dies ist jedoch nicht der Fall, da o_1 und o_8 beide aktuelle Positionen wiedergeben. Da o_1 in DG-Zelle *A* vorhanden ist, wird dessen Position aktualisiert, wohingegen o_4 nicht in der Zelle anzufinden ist und daher neu angelegt wird. Wie in 10.2(a) und 10.2(c) zu sehen ist, wird dabei noch ein neuer MDM-Eintrag mit *MDnum* = 1 für o_4 angelegt, da die DG-Zelle *D* noch einen veralteten Eintrag für o_4 enthält. Nun können o_1 und o_4 aus der MG-Zelle entfernt werden. Angenommen, MG-Zelle *5* erhält kurze Zeit später die beiden Aktualisierungen die Objekte o_6 und o_7 (10.2(b)), so müssen diese in DG-Zelle *D* ausgeschrieben werden. Wenn *D* in den Hauptspeicher eingelesen wird, fällt bei einer Überprüfung des MDM auf, dass die Zelle noch den veralteten Eintrag für o_4 enthält. Dieser kann nun entfernt werden und das

entsprechende *MDnum* wird auf 0 gesetzt, weswegen auch aus dem MDM der Eintrag entfernt werden kann. In DG-Zelle *D* wird o_7 nun aktualisiert und da o_6 noch nicht in ihr vorhanden war, ein neuer Eintrag für o_6 zusammen mit dem entsprechenden MDM-Eintrag angelegt. Dieser letzte Zustand ist in 10.3(a), 10.3(b) und 10.3(c) zu sehen.

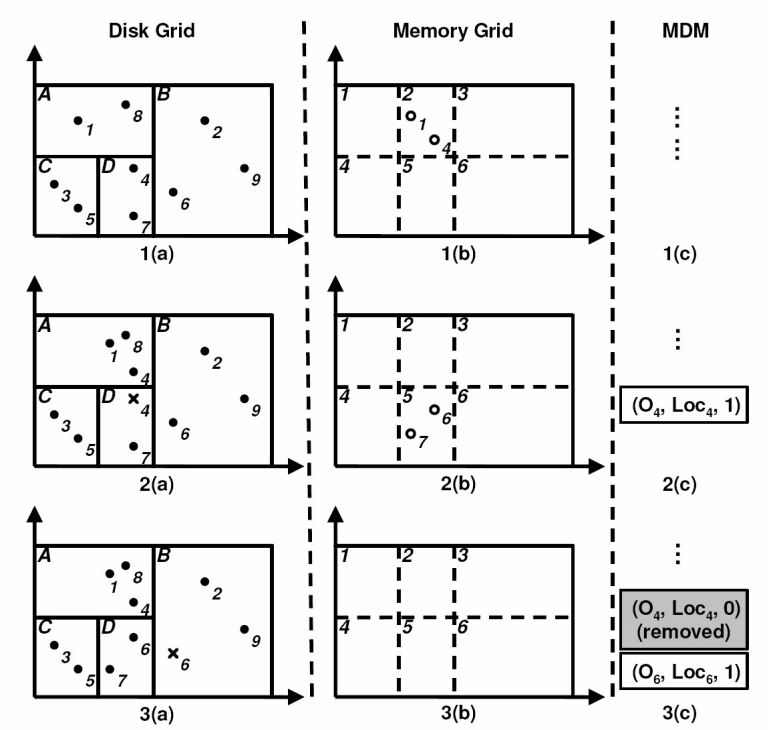

Abb. 10. Schematische Darstellung des LUGrid

Zur Auswertung von Anfragen werden sowohl DG als auch MG herangezogen, da die aktuellsten Positionen entweder noch in MG gepuffert oder schon in das DG ausgeschrieben sein können. Eine Anfrageauswertung wird dabei in zwei Schritten bewerkstelligt. Zuerst wird die Menge der DG und MG bestimmt, die alle in der Anfrage benötigten Objekte enthalten. Der Algorithmus durchsucht dazu alle MG-Zellen, die sich mit dem Anfragegebiet überlappen, nach gepufferten passenden Objektlokationen. Im zweiten Schritt werden die entsprechenden DG-Zellen in den Speicher eingelesen und mit Hilfe des MDM nach aktuellen Positionswerten durchsucht. Danach werden die aktuell gültigen Einträge bestimmt und mit ihnen die Anfrage beantwortet. Da das LUGrid nur die aktuellen Positionen von beweglichen Objekten indexiert, können verständlicherweise keine Anfragen mit Zeitbezug verarbeitet werden.

5 Ausblick

In dieser Ausarbeitung wurde ein MOD-System bisher nur von der Server-Seite aus beschrieben. Jedoch gibt es auch auf der Client-Seite, den beweglichen Objekten, interessante Entwicklungen. Ein bewegliches Objekt übermittelt meist nicht nur seine eigene Position an das Datenbanksystem, sondern bekommt noch kontextbezogene, in diesem Fall ortsabhängige Daten vom System zurückgeliefert. Diese können zum Beispiel räumliche Informationen (Karten) oder die Positionen anderer Objekte sein. Oft ist die Bandbreite der Funkverbindung zum MOD-System jedoch sehr beschränkt oder eine Verbindung ist zeitweise gar nicht möglich, weswegen ein Mechanismus benötigt wird, der den Objekten nur die Daten liefern sollte, die sie voraussichtlich in der nächsten Zeit auch wirklich benötigen. Ein viel versprechender Ansatz, um dies zu bewerkstelligen, ist das so genannte „Hoarding“ [8]. Hierbei werden Daten, die voraussichtlich in der Zukunft von einem Objekt benötigt werden, schon im Voraus übermittelt, nicht benötigte Daten hingegen nicht. Die Auswahl der Daten wird auf Grund der voraussichtlichen zukünftigen Position des Objekts und dessen Geschwindigkeit getroffen. Durch die Übertragung der Daten im Voraus ist eine Versorgung mit kontextbezogenen Informationen somit zumindest für eine gewisse Zeitspanne auch in Gebieten ohne Netzabdeckung möglich.

Im Rahmen dieses Seminars wurden Verfahren vorgestellt, um die mit beweglichen Objekten verbundenen Probleme zu überwinden. In den letzten Jahren wurden dazu viele interessante Ansätze veröffentlicht und, da ortsbezogene Daten die mit am meisten genutzten Kontextinformationen darstellen, ist auch zukünftig mit vielen Veröffentlichungen in diesem Themengebiet zu rechnen. Zumindest die technischen Voraussetzungen für eine praktische Umsetzung der MOD-Systeme sind durch diese Ansätze schon heute gegeben. Die Systeme werden teilweise schon zur Flottenverwaltung in der Logistikbranche eingesetzt und es gibt viele weitere Szenarien, in denen sich die Technik einsetzten ließe. Denkbar ist beispielsweise ein Einsatz in Rettungsleitstellen zur Verwaltung der Einsatzfahrzeuge, um eine bessere Versorgung der Bevölkerung zu ermöglichen. Jedoch lässt sich die Technik auch, ähnlich wie viele andere informationstechnische Entwicklungen, im militärischen Umfeld einsetzen, beispielsweise zur Positionsbestimmung im so genannten „Digital Battlefield“. Es bleibt letztendlich abzuwarten, wann MOD-Systeme in einem großen Maßstab praxisbezogenen umgesetzt werden, so dass eine breite Masse an Endbenutzern das Potential dieser Technik nutzen kann.

Literatur

1. G. Chen, D. Kotz: A Survey of Context-Aware Mobile Computing Research. Dartmouth Computer Science Technical Report TR2000-381, 2000

2. G. Graefe: B-tree indexes for high update rates, ACM SIGMOD Record Vol 35, 2006

3. R. Güting, M. Schneider: Moving Object Databases. Vorlesungsunterlagen der Fernuniversität Hagen (Kurse 1675, 1676, 1677), 2003

4. R. Güting, M. Schneider: Moving Objects Databases, Lecture 2, Modeling and Querying Current Movement, Fernuniversität Hagen, 2002 *(www.informatik.fernuni-hagen.de/pi4/Book%20Moving%20Objects%20Databases/PDF/Lecture%202.pdf)*

5. C. Jensen, D. Lin, B. Ooi: Query and Update Efficient B+-Tree Based Indexing of Moving Objects. Proceedings of the 6th international conference on Mobile data management, 2005

6. C. Jensen, D. Lin, B. Ooi: Query and Update Efficient B+-Tree Based Indexing of Moving Objects. Aalborg University, 2004 *(www.cs.auc.dk/WIM/workshop-07/DanLin.pdf)*

7. B. König-Ries: Anfragen an bewegliche Objekte, Universität Karlsruhe, 2002 *(www.ipd.uka.de/%7Ekoenig/MODIS/beweglicheobjekte.pdf)*

8. U. Kubach, K. Rothermel: A Context-Aware Hoarding Mechanism for Location-Dependent Information Systems, Universität Stuttgart, 2000 *(ftp://ftp.informatik.uni-stuttgart.de/pub/library/ncstrl.ustuttgart_fi/TR-2000-06/TR-2000-06.pdf)*

9. D. Lin, C. Jensen, B. Ooi, S. Saltenis: Efficient Indexing of the Historical, Present, and Future Positions of Moving Objects. Proceedings of the Thirtieth International Conference on Very Large Data Bases, 2004

10. S. Saltenis: Indexing of Moving Objects. Ph.D. Course, March 3-4, 2005, Aalborg University, 2005 *(www.cs.aau.dk/~simas/imo05/imo_phdcourse1.pdf)*

11. A. Sistla, O. Wolfson, S. Chamberlain, S. Dao: Modeling and Querying Moving Objects. Proceedings of the 13 th IEEE Conference on Data Engineering, 1997

12. O. Wolfson, B. Xu, S. Chamberlain, L. Jiang: Moving Objects Databases: Issues and Solutions. Proceedings of the 10th International Conference on Scientific and Statistical Database Management, 1998

13. Wikipedia: Grid file, *http://en.wikipedia.org/wiki/Grid_file*, Stand 06/2007

14. Wikipedia: Space-filling curve, *http://en.wikipedia.org/wiki/Space-filling_curve*, Stand 06/2007

15. X. Xiong, M. Mokbel, W. Aref: LUGrid: Update-tolerant Grid-based Indexing for Moving Objects. Proceedings of the IEEE International Conference of Mobile Data Management, 2006