Sichtenmaterialisierung – Anwendungen und Probleme

Juan C. Fries

1 Einleitung

1.1 Motivation

Data-Warehouse-Systeme greifen in den meisten Fällen auf viele Datenquellen zu. Diese können sehr unterschiedlicher Art sein. Da ein Data Warehouse lokale Kopien von Teilen dieser Daten anlegt, muss es ständig bemüht sein deren Aktualität aufrechtzuerhalten. Hierbei besteht immer die Gefahr von Inkonsistenzen und Anomalien. Diese Arbeit geht näher auf die Probleme näher ein. Der Leser wird in diesem Zusammenhang die Vielfältigkeit der Lösungsansätze und einige Anwendungsbeispiele finden.

1.2 Begriffe

Allem voran müssen wir die in diesem Text verwendeten Begriffe erklären, um ein allgemeines Verständnis der Thematik zu erleichtern.

Unter einer *Sicht* versteht man eine Relation oder Tabelle, die aus einer Basis-Relationen abgeleitet wird und immer wieder neu berechnet werden muss. Die Sicht kann Teile mehrerer Basis-Relationen umfassen.

Sichtenmaterialisierung bedeutet Speichern und lokales Kopieren der Tupel einer Sicht in eine dedizierte Datenbank. Der Zugriff auf die Daten kann hierdurch effizienter erfolgen, da statt auf Sichten auf lokale Tabellen zugegriffen wird. So werden z. B. aufwändige Verbindungen von Relationen vermieden. Des weiteren können in einer Sichtenmaterialisierung Index-Strukturen aufgebaut werden.

Veränderungen an den zugrunde liegenden Basisrelationen können wie bereits angedeutet zu einer Inkonsistenz der Sicht (-enmaterialisierung) führen. Das Aktuellhalten einer Sicht bezüglich solcher Veränderungen wird *Sichtenpflege* oder *View Maintenance* genannt.

2 Anwendungen

- Schnellzugriff

  Jede Anfrage kann auf einen simplen Zugriff einer Sichtenmaterialisierung reduziert werden. Hierdurch wird auch die Belastung des Rechenprozessors und der Festplatten vermindert.

- Data Warehouse

  Informationen können gesammelt werden ohne jede Datenbank in das Data Warehouse kopieren zu müssen. Anfragen werden beantwortet, wobei auf den Zugriff auf die Quell-Datenbank verzichtet werden kann, der beschränkt oder sehr teuer sein kann.

- Chronicle Systems

  Banken, Handels- und Buchhaltungssysteme gehen täglich mit zusammenhängenden Strömen von Transaktionsdaten um. Solche Systeme speichern Reihen von Tupeln in zeitlicher Ordnung ab. Sie können daher sehr groß werden und jenseits jeder Datenbank-Kapazität liegen. Sichtenmaterialisierungen beantworten Anfragen nach den wichtigsten Informationen ohne auf das umfangreiche und schnellwachsende Chronicle System zugreifen zu müssen. Eine Sichtenmaterialisierung könnte z. B. die Kontostände der Kunden liefern.

- Datenvisualisierung

  Visualiserungsanwendungen zeigen Sichten über Daten einer Datenbank. Ändert der Anwender die Sichtendefinition, muss die Anzeige der Daten entsprechend angepasst werden. Mit steigender Erfahrung des Anwenders wächst auch sein Verständnis für die angezeigten Datensammlungen. Dank der Sichtenmaterialisieurng kann das System auf solche interaktiven Änderungen dynamisch reagieren.

- Mobile Systeme

  Mobile Anwendungen verfügen meist über eingeschränkte Möglichkeiten zur Abfrage größerer Datenmengen. Die Geräte auf denen sie laufen verändern ihre Position und werden abhängig von ihrem Ort und der Umgebung Anfragen generieren. Hier ist es sinnvoll, die übertragenen Daten minimal zu halten und nur die Veränderungen neuzuberechnen.

- Integritätsprüfung

  Die meisten statischen Integritätsbedingungen können als eine Menge von Sichten so repräsentiert werden, dass eine nicht-leere Sicht auf eine verletzte Bedingung hinweist. Methoden der View Maintenance können dazu verwendet werden, die Konsistenzbedingungen inkrementell zu überprüfen, wenn Daten einer Datenbank verändert wurden.

- Anfragenoptimierung

  Zur Optimierung beliebiger Anfragen können die Sichten der Sichtenmaterialisierung verwendet werden.

3 Probleme

3.1 View Maintenance Problem

Änderungen an den ausgelesenen Datenquellen können wie weiter oben schon erwähnt zur Inkonsistenz der Sichtenmaterialisierung führen. Um die Aktualität einer Sicht bemüht sich die *Sichtenpflege (engl.: view maintenance)*.

Inkrementelle View Maintenance betrachtet zum Neuberechnen einer Sicht nur die Änderungen in der Datenquelle.

3.1.1 Klassifikation

Anhand der drei in Abb. 1 und gleich genannten vier Dimensionen können die in dieser Arbeit vorgestellten View-Maintenance-Algorithmen klassifiziert werden:

- Information: Die Menge der für die View Maintenance verfügbaren Information. Besteht Zugriff auf die Basisrelationen, Sichtenmaterialisierung und den geänderten Teil der Sicht?
- Modifikation: Mit welchen Modifikationen kann die Sichtenpflege umgehen? Werden Updates als eigenständige Operation oder als ein Lösch- gefolgt von einem Einfügevorgang behandelt? Ändert sich mit den Änderungen der Basisrelationen auch die Sicht?
- Sprache: Wird die Sicht als eine *select-project-join-* (SPJ) oder konjunktive Anfrage ausgedrückt, oder in einer anderen Weise der relationalen Algebra?
- Instanz: Ist der Sichtenpflegealgorithmus für alle Instanzen oder nur für einzelne Instanzen der Datenbank oder nur der Modifikation anwendbar? Die Instanzinformation besteht aus den beiden Typen Datenbank-Instanz und Modifikationsinstanz.

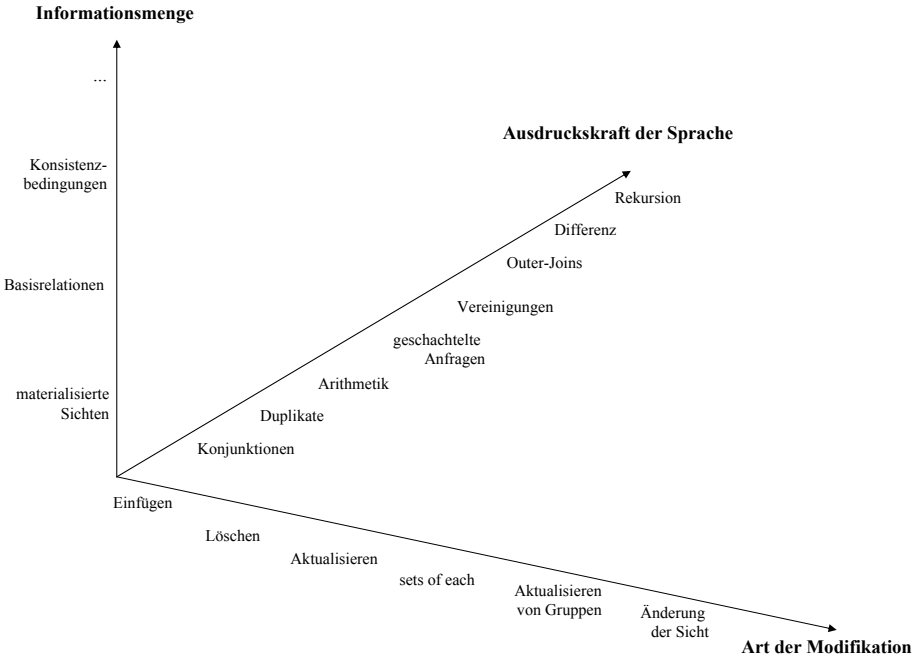

Abbildung 1: Drei Dimensionen des vier-dimensionalen Problemraums

3.1.2 Verwendung vollständiger Informationen

Stehen der Sichtenpflege alle Basisrelationen und Sichtenmaterialisierungen zur Verfügung, spricht man von Sichtenpflege unter Verwendung vollständiger Informationen.

Der Fokus liegt hierbei auf der Entwicklung effizienter Verfahren, Sichten zu pflegen, die in verschieden Sprachen ausgedrückt werden können.

Diese Verfahren kann man in drei Klassen einteilen, für die sich einige Beispiele aufzählen lassen.

1. Rekursive Sichten

- *Deletion and Rederivation (DRed) Algorithm*: DRed nimmt eine zu große Schätzung der zu löschenden Tupel vor und korrigiert diese Liste um die Tupel, für welche alternative Ableitungen existieren.

  Für die Menge der einzufügenden Tupel reicht die Auswertung der teilweise aktualisierten Sichtenmaterialisierungen.
- *Propagation/Filtration (PF) Algorithm*: Der PF-Algorithmus arbeitet analog dem DRed, die Änderungen werden aber für alle Basisrelationen einzeln und schleifenweise berechnet.
- *Kuchenhoff Algorithmus*: Hier werden die Unterschiede nachfolgender Datenbankzustände durch Ableitungen rekursiv neu berechnet. Auch Kuchenhoff arbeitet ähnlich wie DRed, verwirft aber keine mehrfach vorkommenden Ableitungen, die keine Auswirkungen auf die Sicht haben.
- Der *Urpi-Olive Algorithmus* verwendet ein Modell mit Existenzquantoren, um Änderungen der Basisrelationen in geänderte Sichten zu übersetzen.

2. Nicht-rekursive Sichten

   - Der *counting Algorithmus* pflegt mittels eines Zählers über die Ableitungen jedes Tupels der Sicht Einfüge- und Löschoperationen ein.
   - *Algebraic Differencing*: Relationale Ausdrücke berechnen Änderungen der Sicht ohne Redundanz. Für jede Sicht werden die Änderungen mit je eine Sicht für das Einfügen und für das Löschen ermittelt.
   - Der *Ceri-Widom Algorithmus* behandelt ausgewählte SQL-Sichten ohne Duplikate, Aggregation oder Negation getrennt von SQL-Sichten, deren Atribute den Schlüssel der Basisrelation durch eine Funktion bestimmen.
   - Alle Algorithmen der rekursiven Sichten können auch für nicht rekursive Sichten eingesetzt werden.

3. Outer-Join Sichten

   Äußere Verbindungen je einer Relation werden mit der Änderungsmenge der anderen Relation inkrementell gepflegt.

   Zur Sichtenpflege müssen nur die Sichtmaterialisierung und alle Basisrelationen herangezogen werden.

Eine rekursive Sicht verwendet eine Tabelle mehrmals, eine nicht-rekursive verwendet eine Tabelle immer nur einmal.

3.1.3 Verwendung lückenhafter Informationen

Sichten können gepflegt werden mit nur einem Teil der zugrundeliegenden Basisrelationen, die in die Sicht involviert sind.

Hauptaugenmerk liegt dabei in der Erkennung, ob und wie eine Sicht gepflegt werden kann.

- Nullinformation: Muss bestimmt werden, ob eine Änderung an der Basisrelation überhaupt Auswirkungen auf die Sicht hat, spricht man vom *query independent of update* oder auch vom Problem der irrelevanten Aktualisierung. Dies bietet ein weiterhin ein Betätigungsfeld optimierende Algorithmen zu entwickeln.

- Selbstpflege: Eine Sicht heißt *selbstpflegend* (self-maintainable) bezüglich einer Änderungsart (Einfügen, Löschen, Aktualisieren) an einer Basisrelation, wenn die Sicht bei allen Datenbankzuständen sich selbst pflegen kann.

  SPJ-Sichten sind bei Lösch- und Aktualisierungsvorgängen häufig selbstpflegend. Einfügeoperationen können sie nicht berücksichtigen.

- Teil-Referenz: Das *Teil-Referenz*-Sichtenproblem besteht darin, dass die Sicht nur unter Kenntnis eines Teils der Basisrelation und der materialisierten Sicht gepflegt werden muss. Dabei stellen sich zwei interessante Unterprobleme: Einmal wird der Zeitpunkt benötigt, wann die Sicht und alle Relation außer der veränderten Relation verfügbar sind (Chronicle Views); zum anderen wann die Sicht und die modifizerte Relation verfügbar sind (Change-reference Maintainable).

3.2 Offene Probleme

3.2.1 Problemraum

Viele Aspekte blieben noch unberücksichtigt. Der dreidimensionale Problemraum in Abb. 1 wird nicht komplett abgedeckt. Verfahren zur Sichtenpflege unter Verwendung vollständiger Informationen sind in ausreichender Form entwickelt worden. Für unser Problem besteht aber noch ein großer Nachholbedarf, um Algorithmen zu finden, die mit lückenhaften Informationen alle Dimensionen des Problemraums umfassen. Einige Beispiele:

- Verwendung von Informationen über funktionale Abhängigkeiten, mehrfache Sichtenmaterialisierungen, horizontale und vertikale Fragmente der Basisrelationen
- Erweiterung der Definitionssprache um Aggregation, Negation und äußere Verbindungen (Outer-Joins) für alle Instanzen der anderen Dimensionen. Diese Erweiterungen sind für die Verwendung lückenhafter Informationen von besonderer Bedeutung.
- Erkennung von Unterklassen der SQL-Sichten, die sich instanzenunabhängig pflegen lassen.

3.2.2 Problem der Informationsidentifizierung

Das Gegenstück zum Sichtenpflegeproblem bei unvollständiger Information ist das *Problem der Informationsidentifizierung* (Abk.: II, engl.: information identification). Dieses beschäftigt sich mit der Erkennung derjenigen Informationen, die bei einer gegebenen Sicht oder einer gegebenen Menge von Sichten für eine effiziente Sichtenpflege benötigt werden.

Lösungen für die Sichtenpflege mit unvollständigen Informationen können indirekt auf das II-Problem angewandt werden: Man untersuche, ob die gegebene Sicht einem Problem der Sichtenpflege unter Verwendung lückenhafter Informationen entspricht.

Da dies nur unzureichend ist, werden weitere Methoden benötigt, die das II-Problem komplett lösen können.

3.2.3 Weitere ungeklärte Probleme

Ein weiteres Problem ist die Implementierung und Zusammenführung der Sichten in ein Datenbanksystem. In diesem Kontext kommen viele Fragen auf:

- Sollen die Sichtenmaterialisierungen gepflegt werden vor oder nach Abschluss *(commit)* der Transaktion, die die Basisrelation aktualisiert?
- Ist die Sichtenpflege als Bestandteil der Transaktion zu sehen?
- Wird die Sicht am besten bevor oder nachdem die Änderung auf die Basisrelation angewandt wurde gepflegt?
- Muss die Sichtenpflege nach jeder einzelnen Aktualisierung oder erst nach allen Updates erfolgen?
- Soll die Sichtenpflege automatisch (z. B. regelbasierend) oder benutzergesteuert eingeleitet werden?
- Lässt sich ein Kosten-basierendes Modell entwickeln, dass zwischen verschiedenen Lösungswegen abwägt und das günstigste auswählt?
- Bleibt die Frage nach der Komplexität der Sichtenpflege unbeantwortet?

Aber auch die Komplexität der Sichtenpflege bedarf noch der weiteren Erforschung.

4 WHIPS

Als Anwendungsbeispiel lässt sich das *WHIPS*-Projekt aufführen. WHIPS (Data WareHouse Prototype at Stanford)[1] ist ein mittlerweile abgeschlossenes Projekt der Stanford University zur Entwicklung von Algorithmen und Werkzeugen, die der Erstellung und Pflege von Data Warehouses dienen sollen.

4.1 Abgrenzung vom herkömmlichen View Maintenance Problem

Die Forschungsgruppe um WHIPS gibt zu, dass wenn man die Daten in einem Data Warehouse als Sichtenmaterialisierung betrachtet, die zuvor angegeben Algorithmen zur Sichtenpflege angebracht sein könnten. Sie stellen aber klar, dass sich die Sichtenpflege vom Data Warehousing in zwei entscheidenden Punkten unterscheidet:

1. Die Informationsquellen können heterogen und
2. autonom sein.

Die Informationsquellen kooperieren nicht zwangsläufig mit der Sichtenverwaltung. Die Überwachung kann somit sehr problematisch werden. Dies tangiert den Bereich der aktiven Datenbanken, der sich mit der Überwachung individueller Datenbanken bezüglich Änderungen beschäftigt.

4.2 Architektur

Der Aufbau eines Data Warehouses im WHIPS-Stil lässt sich der Abb. 2 entnehmen. Anwendungen der Dienstnehmer stellen Anfragen an das Data Warehouse. Um die Dienstnehmer mit den aktuellsten gewünschten Informationen beliefern zu können, erfolgt ein Zusammenspiel von Data Warehouse, dem Integrator und den Monitoren, von denen jeweils einer für eine der Informationsquellen zuständig ist.

Im endgültigen WHIPS-System mussten weitere Komponenten hinzugefügt, um die Ziele des Projekts erreichen zu können.

[1] http://www-db.stanford.edu/warehousing

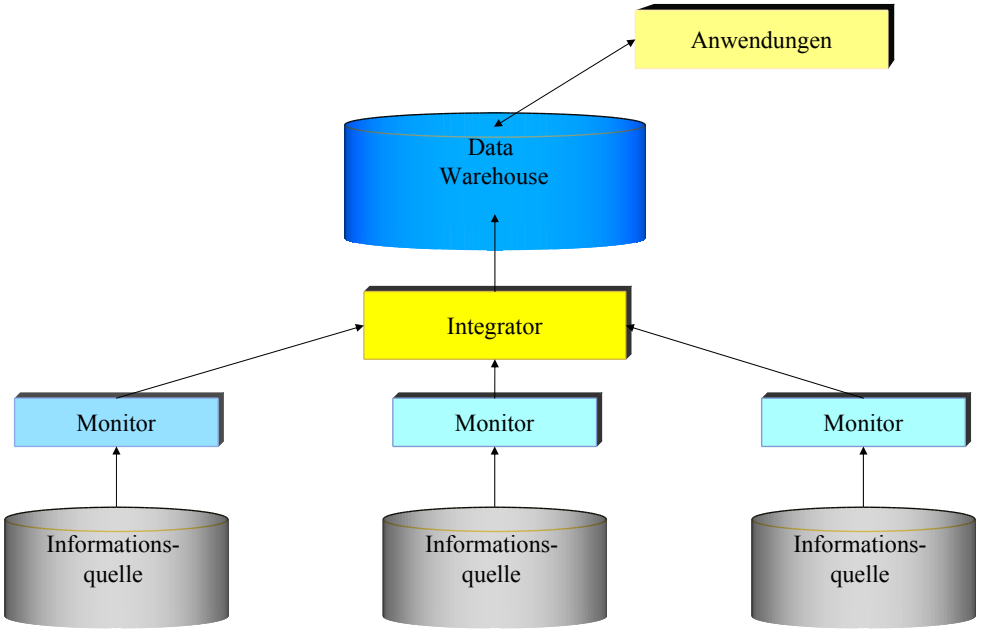

Abbildung 2: Aufbau eines Data Warehouses

4.2.1 Monitore

In die Verantwortung der Monitore gehört die Erkennung und Bekanntmachung von Veränderungen im Datenbestand einer Informationsquelle. Bei Änderungen benachrichtigt der Monitor den Integrator über diese Änderungen.

Eine Möglichkeit zur Realisierung ist die Verwendung von Auslöseprozessen (Trigger), die bei Veränderungen des Datenbestands aktiv werden und die Benachrichtigung versenden. Die andere Möglichkeit besteht in der Prüfung der Protokolldateien (update logs), die die Aktualisierungen mitschreiben, auf wesentliche Einträge.

Systeme mit Altlast *(legacy systems)* verfügen leider weder über Trigger- noch Loggingfunktionen. In diesem Fall muss man sich einiger Hilfskonstruktionen behelfen. Das WHIPS-Team schlägt zwei Varianten vor:

1. Jede Anwendung, die Zugriff auf die Daten in den zugrundeliegenden Informationsquellen haben soll, muss so modifiziert werden, dass sie bei jeder Änderung von sich aus Mitteilungen an den Monitor sendet.

2. Die Urdaten der Informationsquellen werden von einem extra hierzu geschriebenen Programm in eine Datei gekippt. Legacy Systems verfügen über diese Speicherfunktion meist als Funktion zur Datensicherung. Anschließend meldet der Monitor dem Integrator Veränderungen dieser Datei als Datenbank-Änderungen. Lösungen zum Auffinden solcher Veränderungen sind in der Forschung im Zusammenhang mit dem *snapshot differential problem* bekannt und werden in dieser Arbeit daher nicht weiter ausgeführt.

4.2.2 Integratoren

Wie schon oben erwähnt, ist der Integrator das Glied im Data-Warehouse-Konstrukt, das über Veränderungen in den Datenbeständen informiert wird.

Die Hauptaufgabe des Integrators besteht darin, die Sichtenpflege wesentlich zu vereinfachen, indem er ermittelt welche Modifikationen welche Sicht betreffen.

Für alle Änderungstypen werden Regeln definiert, die festlegen welche Schritte der Integrator ergreifen muss, um die entsprechenden Änderungen mit dem Data Warehouse korrekt zusammenzuführen.

Probleme die auftreten können, wenn der Integrator in diesem Zusammenhang weitere Daten benötigt, werden im nächsten Abschnitt behandelt.

Wünschenswertes Ziel wäre es eine Sprache auf sehr hoher Ebene zu entwickeln, in der man genau angeben kann wie die relevanten Daten in das Data Warehouse eingebunden werden sollen. Dies würde ein Höchstmaß an Flexibilät und Konfigurierbarkeit ermöglichen. Selbst Veränderungen der Meta-Daten von Datenbank oder Warehouse würden so erlaubt.

4.3 Warehouse Update-Anomalien

Betrachtet man ein Data Warehouse als das bloße Kopieren von Sichtenmaterialisierungen, vergisst man, dass in einem Warehouse die Quelldatenbanken nicht alle zum selben System gehören müssen. Sie können völlig unabhängig voneinander entwickelt und aufgebaut worden sein, ihre Methoden und Formate unterschiedlich sein und Sichtenverwaltung und Datenbank voneinander entkoppelt arbeiten.

Wenn eine Informationsquelle dem Integrator eine Benachrichtung übermittelt, weiss die Quelle eventuell nicht, dass der Integrator zur Verarbeitung dieser Änderung weitere Daten aus dieser oder anderen Quellen benötigt. Diese Aufgabe muss der Integrator selbst übernehmen. Während der Integrator ermittelt, welche Daten er noch abfragen muss, kann es dazu kommen, dass Daten der Informationsquelle sich zwischenzeitlich erneut geändert haben. Die inzwischen überholte Antwort auf die Anfrage des Integrators kann so zu einer inkorrekten Sicht führen. Man spricht hierbei von *warehouse update anomalies*.

Glücklicherweise lassen sich diese Update-Anomalien verhindern. Die Informationsquelle muss über keine eigene Sichtenverwaltung verfügen. Sie beantwortet einfach nur Anfragen des Integrators und verständigt ihn bei bestimmten Änderungen. Das Konzept des Data Warehouses erfordert zudem, dass auf die auszuwertenden Relationen keine Sperre gesetzt wird.

Diese Lösungen können daher herangezogen werden:

- Neuberechnen der Sicht:

  Der Integrator kann die Sicht entweder bei jeder Änderung an der Informationsquelle oder periodisch in bestimmten Zeitabständen neuberechnen. Diese Lösung ist meist Zeit- und Ressourcen-aufwändig.

- Lokale Kopien aller involvierten Daten:

  Werden alle für die Sicht relevanten Daten der Informationsquellen auch lokal als Kopien im Data Warehouse gespeichert, können die Anfragen des Integrators lokal beantwortet werden und es kommt zu keinen Update-Anomalien. Verschwendeter Speicherplatz und enormer Overhead bilden die Kehrseite.

- Eager Compensating Algorithmus:

  Der Integrator setzt Anfragen ab, die den Effekt nebenläufiger Updates kompensieren. Dadurch bleibt der Overhead minimal, die Daten müssen weder lokal gehalten noch ständig neu berechnet werden. Der Eager Compensating Algorithmus ist in der Literatur ausführlich beschrieben.

5 Zusammenfassung

Einige Probleme der Sichtenmaterialisierung wurden erörtert und Anwendungsmöglichkeiten aufgezeigt. Einige Probleme konnten schon gelöst werden, viele Aspekte sind aber noch unberücksichtigt geblieben.

Als konkretes Beispiel diente das Data-Warehouse-Projekt WHIPS, das für seine Belange ausreichende Lösungen präsentieren kann. Die Forschungsarbeit auf diesem Gebiet kann aber noch lange nicht als abgeschlossen gelten.

So konnten in Bezug auf Konsistenzwahrung weitere Fortschritte erreicht werden. Deren Behandlung ist so bedeutsam und umfangreich, dass eine andere Arbeit sich damit beschäftigen musste.

Data-Warehouse- und mobile Systeme wurden in der vorliegenden Arbeit bisher nur als getrennte Anwendungen betrachtet. Interessante Entwicklungen und Ergebnisse verspricht das Projekt zur Stärkung der SelbstOrganisationsfähigkeit im Verkehr durch I+K-gestützte Dienste (OVID[2]) an der Universität Karlsruhe. Bedenkt man, dass Verkehr auch Bewegungen von diversen Subjekten mit Fortbewegungsmitteln aller Art zu Lande, zu Wasser und in der Luft bedeutet, eröffnet sich ein weites Betätigungsfeld.

[2] http://www.ovid.uni-karlsruhe.de

Literatur

[GuMu95] Ashish Gupta und Inderpal Singh Mumick. Maintainance of Materialized View: Problems, Techniques, and Applications. *Bulletin of the Technical Committee on Data Engineering, IEEE* 18(2), Juni 1995, S. 3–18.

[GuMu99] Ashish Gupta und Inderpal Singh Mumick (Hrsg.). *Materialized Views: Techniques, Implementations, and Applications*. MIT Press. 1999.

[HGMWL+95] Joachim Hammer, Hector Garcia-Molina, Jennifer Widom, Wilburt Labio und Yue Zhuge. The Stanford Data Warehousing Project. *Bulletin of the Technical Committee on Data Engineering, IEEE* 18(2), Juni 1995, S. 40–47.

[WGLZ+96] J. L. Wiener, H. Gupta, W. J. Labio, Y. Zhuge, H. Garcia-Molina und J. Widom. A System Prototype for Warehouse View Maintenance. *ACM Workshop on Materialized Views: Techniques and Applications*, Juni 1996, S. 26–33.