

Technische Grundlagen für eine laufzeitadaptierbare Transaktionsverwaltung

Florian Irmert, Christoph P. Neumann, Michael Daum,
Niko Pollner, Klaus Meyer-Wegener

Lehrstuhl für Informatik 6 (Datenmanagement), Department Informatik
Friedrich-Alexander-Universität Erlangen-Nürnberg
{florian.irmert, christoph.neumann, md, kmw}@informatik.uni-erlangen.de

Abstract:

Auf spezielle Anwendungen zugeschnittene Datenbankverwaltungssysteme (DBMS) erfordern in den meisten Fällen Individuallösungen, welche einen erhöhten Entwicklungs- und Wartungsaufwand mit sich bringen. Es fehlen modulare DBMS-Architekturen als konfigurierbare Standardlösung, die beliebig schlanke als auch umfangreiche Architekturen ermöglichen. Die Transaktionsverwaltung als querschnittender Belang durchdringt in herkömmlichen DBMS die Architektur und verhindert die Modularisierung. In dieser Arbeit wird die Transaktionsverwaltung aus einem DBMS herausgelöst und durch eine eigenständige, wiederverwendbare Komponente realisiert. Die Architektur der gekapselten und wiederverwendbaren Transaktionsverwaltung wird dokumentiert, sowie die aspektorientierte Verknüpfung zwischen reduziertem DBMS und separater Transaktionsverwaltung beschrieben. Als Grundvoraussetzung für zukünftige, sich selbstverwaltende DBMS kann die Transaktionsverwaltung zur Laufzeit dynamisch hinzugefügt und entfernt werden. Die prototypische Implementierung wird an Hand von SimpleDB in Verbindung mit einem dynamischen AOP Framework erläutert.

1 Einleitung

Existierende Datenbanksysteme sind in der Regel monolithische Systeme, die nicht für jedes Anwendungsgebiet optimal geeignet sind. Um Datenbanken bspw. in eingebetteten Systemen einzusetzen, ist ein möglichst geringes Speicherprofil von entscheidender Relevanz. Hierfür optimierte Speziallösungen haben jedoch einen erhöhten Entwicklungs- und Wartungsaufwand zur Folge. Eine nachhaltige Lösung muss darin bestehen, die Anpassungsfähigkeit und die schlanke Architektur einer möglichen Individualentwicklung mit dem umfassenden Funktionsangebot traditioneller Datenbanksysteme zu vereinen.

In eingebetteten Systemen ist oft auf Basis einer spezifischen Anwendungssemantik oder wegen einfachen und leicht zu synchronisierenden Anwendungen die Transaktionsverwaltung (TAV) ungenutzter Ballast. Ein schlankes Datenbankverwaltungssystem (DBMS) ohne TAV mit erhöhtem Durchsatz und besserem Antwortzeitverhalten ist für dieses Anwendungsgebiet optimal. Da nicht auszuschließen ist, dass das Anwendungsgebiet in seiner Komplexität mit der Zeit derart zunimmt, dass später eine TAV benötigt wird, soll die

Ergänzung des DBMS um den Transaktionsaspekt möglichst nahtlos stattfinden können.

Eine Herausforderung ist die Modifikation eines modularen Datenbanksystems zur Laufzeit. Gray postulierte „if every file system, every disk and every piece of smart dust has a database inside, database systems will have to be self-managing, self-organizing, and self-healing“ [Gra04]. Der Bedarf für dynamisches Austauschen und Aktualisieren von Modulen zur Laufzeit als relevante Forschungsaufgabe wurde bereits in den 70er und 80er Jahren erkannt [Fab76, SF89] und ist bis heute eine Herausforderung. In unserem Kontext bedeutet dies die Ergänzung des DBMS auf dem Endgerät um die TAV ohne eine erneute Auslieferung des DBMS und ohne einen damit verbundenen Neustart.

Das Projekt CoBRA DB (**C**omponent **B**ased **R**untime **A**daptable **D**ata**B**ase) stellt eine moderne Referenzmodularisierung zur Verfügung [IDMW08]. Insbesondere die architektonische Kapselung der TAV stellt eine der größten Herausforderungen bei der Modularisierung eines DBMS dar, da die TAV ein rein konzeptionellen Systembaustein ist, der als querschnittender Belang viele Systembausteine eines DBMS durchdringt.

Ziel dieser Arbeit ist es, eine Referenzarchitektur und prototypische Implementierung vorzustellen, in der die TAV als streng eigenständiger Aspekt gekapselt ist, so dass sie auch in anderen DBS-Entwicklungen als Baustein eingesetzt werden kann. Zur Realisierung wird dynamische aspektorientierte Programmierung (d-AOP) [CS04] verwendet, eine AOP-Infrastruktur mit Unterstützung von Modifikationen zur Laufzeit. Die Modularisierung erfolgt im vorgestellten Prototypen auf Basis der SimpleDB [Sci07], die in die klassischen Schichten eines relationalen DBMS nach [Här05] gegliedert ist.

2 Verwandte Arbeiten

In diesem Kapitel wird dynamische AOP (d-AOP) kurz vorgestellt sowie andere Arbeiten im Bereich der modularen DBMS abgegrenzt.

2.1 Dynamische aspektorientierte Programmierung

Das Einweben von Code eines Aspekts, sog. Advices, auf Basis von Pointcuts kann grundsätzlich zu drei verschiedenen Zeitpunkten geschehen: Beim Übersetzungsvorgang (*compile time weaving*), zum Ladezeitpunkt der Klasse (*load time weaving*) oder zur Laufzeit (*runtime weaving*). Kann Aspektcode nach dem ersten Einweben verändert werden, wird von *dynamic AOP* gesprochen. In dieser Arbeit wird *JBoss AOP* verwendet, bei dem zur Ladezeit an denjenigen Stellen, an denen zur Laufzeit Advices ausgeführt werden können, sogenannte Hooks eingewebt werden [CS04].

2.2 Modulare Datenbankverwaltungssysteme

Vor- und Nachteile aspektorientierter Methoden in der Entwicklung von DBMS werden in [TSH04] diskutiert. Dabei liegt beim Refactoring des modularen DBMS *Berkeley DB* unter Verwendung von AOP der Schwerpunkt auf Wartbarkeit. *Berkeley DB* dient ebenfalls als Grundlage des in [Käs07] vorgestellten aspektorientierten Refactorings mittels *AspectJ*. Die Umgestaltung konzentriert sich nicht auf die TAV, sondern auf feingranulärere Systemteile, da in erster Linie die Anwendbarkeit von AOP zur Implementierung von Features diskutiert wird.

In [PLKR07] wird die Entwicklung eines konfigurierbaren Transaktionsmanagements vorgestellt. Für die Umsetzung kamen *Aspectual Mixin Layers* zum Einsatz, eine Verbindung aus AOP und FOP (Feature Oriented Programming). Schwerpunkt bildet die Konfigurierung auf Basis identifizierter Merkmale innerhalb des Transaktionsmanagements. Im Rahmen des *COMET DBMS*-Projekts [NTNH03] erfolgt die Entwicklung eines komponentenbasierten DBMS unter Verwendung von aspektorientierter Programmierung.

Die vorliegende Arbeit unterscheidet sich von den oben genannten vor allem in zwei Bereichen: Zum einen konzentriert sie sich auf das Gebiet der Transaktionssicherung und gibt einen Architekturvorschlag für deren DBMS-unabhängige Implementierung. Zum anderen steht bei diesem Design die Möglichkeit des Hinzufügens und Entfernens des für die Transaktionssicherung zuständigen Codes zur Laufzeit im Vordergrund.

3 Anforderungen

Im Folgenden werden die grundsätzlichen Anforderungen an ein DBMS beschrieben, dessen TAV mit Hilfe von d-AOP angebunden werden soll. Des Weiteren wird begründet, dass Schlüsselwörter der Transaktionsunterstützung auch ohne Existenz der TAV vorbereitend durch die DBS-Nutzer und -Programmierer in SQL verwendet werden müssen.

3.1 TAV-Unabhängigkeit des DBMS

Es gibt mehrere Gründe, die eine Forderung nach Unabhängigkeit des DBMS von der TAV bedingen: 1) Die TAV soll nur bei Bedarf hinzugefügt werden und auf Wunsch auch wieder entfernt werden können. 2) Die Entwicklung und die Wartung des restlichen Systems werden durch direkte Berücksichtigung der TAV stark erschwert, da sich diese quer durch alle Schichten zieht. Bei einer nachträglichen Änderung an der TAV sind somit möglicherweise Änderungen an vielen Klassen notwendig, die keinen direkten Bezug zur Aufgabe der Transaktionssicherung haben.

3.2 DBMS-Unabhängigkeit der TAV

Auch in Bezug der Abhängigkeit der TAV vom DBMS ist aufgrund verschiedener Forderungen eine möglichst lose Kopplung anzustreben: 1) Änderungen am zugrundeliegenden System sollen nur möglichst wenige Änderungen an der TAV nach sich ziehen. 2) Der mögliche Einsatz einer einmal entwickelten TAV in verschiedenen Datenbanksystemen verringert sowohl den Entwicklungs- als auch Wartungsaufwand. 3) Bei einem konfigurierbaren System können die Implementierungen der einzelnen Schichten nach Bedarf ausgetauscht werden. Es ist deshalb anzustreben, die TAV so zu entwerfen, dass sie von einem solchen Austausch möglichst wenig betroffen ist.

Im Gegensatz zur TAV-Unabhängigkeit lässt sich DBMS-Unabhängigkeit nur bis zu einem gewissen Grad erreichen. Die Anbindung ist zwangsläufig spezifisch für ein bestimmtes System. Zur Durchführung seiner Aufgaben benötigt insbesondere das für Recovery zuständige Teilsystem Zugriff auf interne Strukturen der Datenbank. Die Wartbarkeit lässt sich hier mit Hilfe einer günstigen Architektur weitestgehend erhalten, indem der für die Anbindung bedeutsame Code in wenigen Klassen gekapselt und von der eigentlichen Logik der TAV getrennt wird.

3.3 Vorbereitende Verwendung transaktionaler Schlüsselwörter

Aus Sicht des DBS-Nutzers muss stets Isolation gewährleistet sein. Im Einbenutzerbetrieb gilt dies immer, für Mehrbenutzerbetrieb wird im Allgemeinen eine TAV benötigt. Mehrbenutzerbetrieb ohne TAV ist möglich, wenn die Anwendungen per Anwendungssemantik isoliert sind.

Für den DBS-Nutzer und -Programmierer beginnt mit jeder DBS-Verbindung eine logische Folge von Operationen; setzt er explizit SQL-Schlüsselwörter der Transaktionsunterstützung ein, beginnt für ihn anschließend eine neue logische Folge. Auf Systemseite wird dabei eine Transaktion beim Aufbau einer Verbindung und nach jedem *commit* und *abort* begonnen, entsprechend der *SQL*-Norm und dem Standardverhalten der *JDBC*-API. Ohne TAV werden mit Schlüsselwörtern der Transaktionsunterstützung nur logische Abschnitte markiert: Ein *commit* stellt ohne TAV insbesondere die Dauerhaftigkeit nicht notwendigerweise sicher, ein *abort* ist ohne TAV nicht möglich und dem Anwender muss ein Fehler mitgeteilt werden.

Im Allgemeinen kann die Datenbankzugriffsschicht auf Anwendungsseite zur Laufzeit nicht geändert werden. Da das DBMS die TAV zumindest potentiell in der Zukunft zur Verfügung stellt, müssen die Benutzer und Programmierer ihre logischen Operationsfolgen innerhalb von Sitzungen von Beginn an für den allgemeinen Mehrbenutzerbetrieb vorbereiten, indem sie *commit* verwenden. Verzichten sie darauf, muss ihnen klar sein, dass später ein DBMS mit TAV aus einer Sitzung eine einzige Transaktion ableiten wird. Besonders in eingebetteten Systemen laufen Anwendungen und ihre Sitzungen oftmals ohne Terminierung und würden in nicht-terminierenden Transaktionen über die Zeit Sperren akkumulieren, die niemals freigegeben werden. Als Vorbereitung einer dynamischen

Zuschaltung der TAV ist daher der Einsatz von *commit*-Anweisungen zur Strukturierung in logische Folgen notwendig.

4 Architektonische Kapselung der TAV

In der ursprünglichen SimpleDB wird die Ablaufverfolgung mit Hilfe eines Objekts vom Typ `Transaction` realisiert, das eine einzelne Transaktion repräsentiert. Diese Instanz wird, beginnend bei einer in der obersten Schicht eingehenden Anforderung, bei jedem Methodenaufruf als Parameter weitergereicht und durchläuft so alle Schichten, wodurch der Transaktionskontext überall verfügbar ist.

Zur Transaktionssicherung werden bei Zugriffen auf den Systempuffer, statt direkter Aufrufe an diesem, entsprechende Methoden des `Transaction`-Objekts verwendet, welche die zur Sicherstellung der Transaktionseigenschaften notwendigen Maßnahmen durchführen. Durch diese Technik entsteht eine feste Kopplung zwischen allen Klassen, die Zugriff auf den Systempuffer benötigen, und der TAV. Im Prototyp wurde zunächst jeglicher Transaktions-Code manuell aus der SimpleDB entfernt. Das Package-Diagramm in Abbildung 1 gibt einen Überblick über die Architektur der entwickelten TAV.

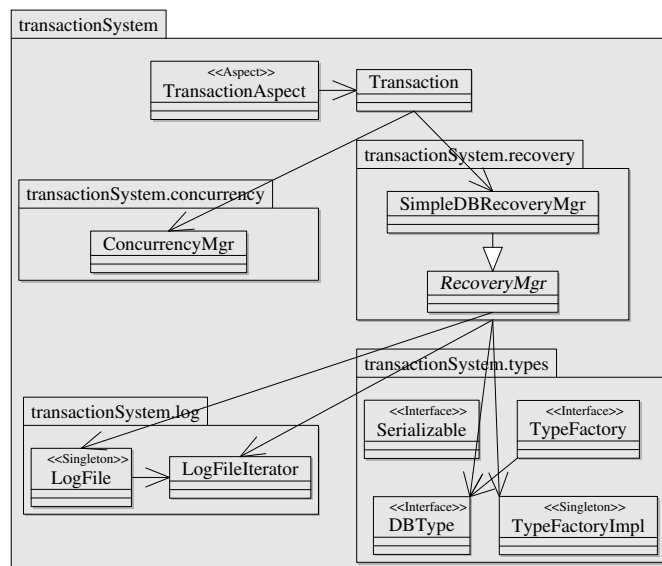


Abbildung 1: Überblick über die Architektur der TAV

4.1 Anbindung der TAV an das Grundsystem

Die äußere Schnittstelle der TAV, über die sie an die Vorgänge im zugrundeliegenden DBMS mittels Methoden wie z.B. `commit()`, `rollback()`, usw. angekoppelt wird, ist in der Klasse `Transaction` realisiert. Eine Instanz repräsentiert dabei genau eine Transaktion.

Die Klasse `TransactionAspect` sorgt für die Anbindung der Objekte vom Typ `Transaction`. Sie ist DBMS-abhängig und enthält `Advices`, die an allen relevanten Stellen des grundlegenden Systems die Methoden der betroffenen `Transaction`-Objekte aufrufen.

4.2 Innere Architektur der TAV

Die Klasse `LogFile` stellt die für das Recovery-System benötigten Funktionen zum Logging bereit. Sie bietet Methoden zum Schreiben von allgemeinen Daten in eine serielle Datei und unterstützt über einen entsprechenden Iterator das Auslesen. Da diese Klasse keine Annahmen über den Inhalt oder interne Strukturen macht, ist sie universell einsetzbar, unabhängig von der konkreten Ausprägung des Recovery-Mechanismus.

Die Klassen im Package `concurrency` stellen den Concurrency-Manager der TAV dar. Als Schnittstelle zur Klasse `Transaction` dient `ConcurrencyMgr`, von dem jeweils eine Instanz für eine Transaktion zuständig ist. Die Implementierung ist unabhängig von dem zugrundeliegenden DBMS. Das Sperrverfahren kann als Teilmodul ausgetauscht werden.

Im Package `recovery` befinden sich die für das Recovery Management zuständigen Klassen. Da der Recovery-Manager auf die Speicherstrukturen des DBMS zugreifen muss, um Logging und Recovery durchführen zu können, wurde das *Template Method Pattern* eingesetzt, um eine Unabhängigkeit der grundlegenden Recovery-Prozedur zu erreichen. Die Recovery-Strategie ist in der prototypischen Realisierung eine reine Undo-Recovery und kann als Teilmodul durch eine Redo- und Undo-Recovery ausgetauscht werden.

Um eine generische Repräsentation von Daten zu ermöglichen, unabhängig von den im zugrundeliegenden System hierfür eingesetzten Typen, wurde eine Hierarchie spezieller Klassen im Package `types` entworfen. `Serializable` stellt eine einfache Schnittstelle für Klassen dar, die eine Serialisierung und Deserialisierung unterstützt. Klassen, die einen Datentyp des zugrundeliegenden DBMS repräsentieren, implementieren die Schnittstelle `DBType`, die neben den Methoden von `Serializable` noch die Methode `getTypeId()` besitzt. Über diese kann eine Identifikationsnummer abgerufen werden, die für einen bestimmten Datentyp eindeutig ist. In der Logdatei können so die Identifikationsnummer des Datentyps und danach die Daten in serialisierter Form gespeichert werden.

4.3 Verfolgung der Transaktionen

Dynamische AOP bietet keine Möglichkeit, den Kontrollfluss innerhalb einer Anwendung zu überwachen. Die Verfügbarkeit von Kontrollflussinformation ist aber bei der Realisierung einer TAV unabdingbar, da die Aufrufe in den einzelnen Komponenten der jeweiligen Transaktion zugeordnet werden müssen.

In der ursprünglichen SimpleDB wurde der Transaktionskontext im `Transaction`-Objekt durch alle Schnittstellen erreicht, was bei einer echten Trennung der TAV vom DBMS eliminiert werden muss. Die grundlegende Problematik, einen transaktionalen Kontext zu bestimmen ohne eine enge Kopplung der TAV mit dem DBMS zu verursachen, wurde bereits in [Käs07] diskutiert, jedoch ohne eine für die Zwecke der vorliegenden Arbeit zufriedenstellende Lösung aufzuzeigen.

Eine Möglichkeit zur Verfolgung des Kontrollflusses stellen die Threads des Systems dar, da diese den Kontrollfluss repräsentieren. Die Ermittlung des transaktionalen Kontexts durch die TAV kann mit Hilfe von Tabellen realisiert werden, welche die Zuordnungen zwischen JDBC-Connection, SQL-Statement, Transaktion und Thread verwalten. Die Tabellen werden durch die TAV vollkommen autonom verwaltet, die notwendigen Advices werden in das DBMS eingebracht.

Der Mechanismus der Transaktionskontextermittlung wird in Abbildung 2 dargestellt. Grundlage des Beispiels ist die Ausführung einer SQL Update-Anweisung. Die mit AOP gekennzeichneten Aufrufe sind aspektorientierte Aufrufe von Advices der TAV, die durch entsprechende Pointcutdefinitionen ausgelöst werden. Die Sequenz beginnt mit dem Aufruf von `createStatement()` durch den Client, mit dem Ziel eine Referenz auf ein `RemoteStatementImpl`-Objekt zu erhalten. Das `RemoteConnectionImpl`-Objekt repräsentiert die bereits bestehende Verbindung.

An der mit 1 bezeichneten Stelle wird vor der Rückkehr des `createStatement`-Aufrufs die Zuordnung der neuen `RemoteStatementImpl`-Instanz zum `RemoteConnectionImpl`-Objekt in einer `statement-connection` Tabelle abgelegt. Um ein Update durchzuführen, ruft der Client im Beispiel die Methode `executeUpdate()` auf.

Bei Stelle 2 wird über den in 1 erzeugten Tabelleneintrag die zugehörige Verbindung ermittelt und anhand einer weiteren `connection-transaction` Tabelle geprüft, ob für diese Verbindung gerade eine Transaktion läuft. Im Rahmen dieses Beispiels enthält die Tabelle noch keinen Eintrag. Daher wird an Stelle 2 innerhalb der TAV ein neues `Transaction`-Objekt erzeugt und somit eine neue Transaktion gestartet. Anschließend wird die `connection-transaction` Tabelle um die Zuordnung zwischen Verbindung und neuer Transaktion ergänzt. Außerdem erfolgt ein Eintrag in einer `thread-transaction` Tabelle, der dem aktuellen Thread die ermittelte bzw. neu erzeugte Transaktion zuordnet.

Die drei verwendeten Verwaltungstabellen haben eine unterschiedliche Intention: Die `thread-transaction` Tabelle liefert direkt und effizient bei allen folgenden Satzoperationen des ausgeführten SQL-Statements den transaktionalen Kontext. Die `connection-transaction` Tabelle und `statement-connection` Tabelle werden bei neu eintreffenden SQL-Statements benötigt, um einen neuen Thread einer bereits laufenden Transaktion zuzuordnen.

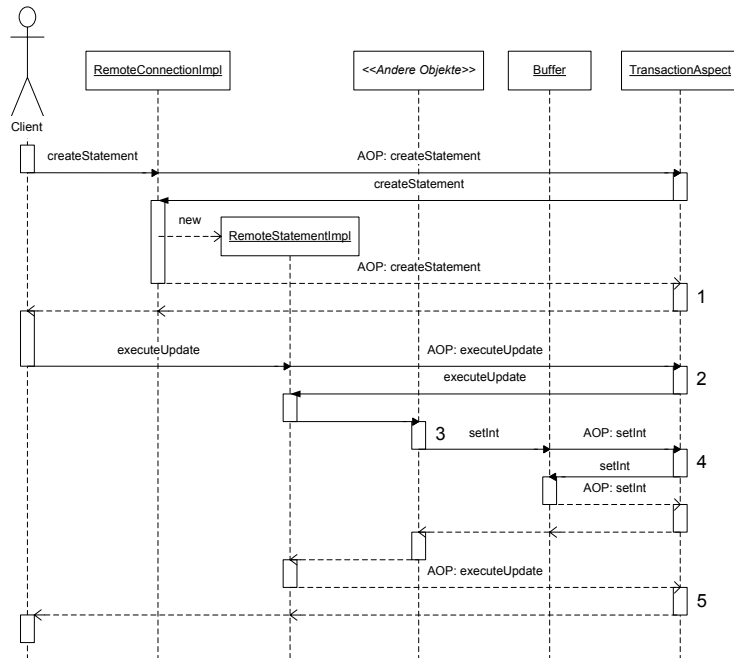


Abbildung 2: Sequenzdiagramm der Transaktionsverfolgung (leicht vereinfacht)

Nach dem Anlegen der Verwaltungsinformation veranlasst der Advice die Ausführung der eigentlichen `executeUpdate()`-Methode. Sie wird in ihrem Verlauf Satzoperationen aufrufen (Stelle 3), von denen letztendlich ein Aufruf an den Systempuffer (Buffer) abgesetzt wird, der Maßnahmen durch die TAV erfordert. Im Diagramm ist dies beispielhaft für `setInt()` dargestellt: Bei Stelle 4 wird über die `thread-transaction` Tabelle das für diesen Aufruf passende `Transaction`-Objekt ermittelt. Die TAV veranlasst über dieses Objekt die notwendigen Maßnahmen wie Sperranforderung und Logging.

An der Stelle 5 schließlich wird vor der Rückkehr des `executeUpdate()`-Aufrufs an den Client die Zuordnung von Thread zu Transaction wieder aus der entsprechenden Tabelle entfernt.

5 Zusammenfassung

In dieser Arbeit wurde die prototypische Implementierung einer mittels d-AOP angebotenen TAV vorgestellt. Als Grundlage diente ein um die TAV reduziertes schlankes DBMS auf Basis der SimpleDB, deren Funktionsumfang vollständig erhalten wurde. Zielumgebungen sind eingebettete Systeme, in denen einerseits ein schlankes Speicherprofil der Softwarebausteine besondere Relevanz hat, aber andererseits die Anpassbarkeit an eine

zukünftige Anwendungsumgebung gefordert wird.

Die TAV wurde als autonomes und DBMS-unabhängiges Modul realisiert, das mit Hilfe eines Transaktionsaspekts an das DBMS angeschlossen ist. Die beschriebene Architektur optimiert die DBMS-Unabhängigkeit der TAV, indem die DBMS-abhängigen Anteile ausschließlich in Form von Pointcuts und Advices repräsentiert sind, als Anschlussstellen an das DBMS. Abhängigkeiten der TAV-Subsysteme für Synchronisation, Logging und Recovery zum zugrunde liegenden DBMS wurden auf ein Minimum reduziert. Der Einsatz eines d-AOP Frameworks ermöglicht es, das Hinzu- und Abschalten der TAV zur Laufzeit vorzunehmen.

Das übergeordnete Ziel ist es, eine modulare DBMS-Architektur und einen „DBS-Baukasten“ zu gewinnen, der umfassende Anpassungsfähigkeit an beliebige Anwendungsumgebungen unterstützt und dabei sowohl beliebig schlanke Architekturen als auch umfangreiche Architekturen ermöglicht. Diesem Ziel sind wir um den autonomen Baustein der Transaktionsverwaltung nähergekommen. Zukünftige Aufgabe ist die exakte Definition der transaktionalen Semantik sowie die Realisierung einer Zustandsmigration bei Adaption der TAV zur Laufzeit.

Literatur

- [CS04] R. Chitchyan und I. Sommerville. Comparing Dynamic AO Systems. In *Dynamic Aspects Workshop (DAW 2004)*, Lancaster, England, März 2004.
- [Fab76] R. S. Fabry. How to design a system in which modules can be changed on the fly. In *2nd International Conference on Software Engineering (ICSE 1976:)*, Seiten 470–476, Los Alamitos, CA, USA, Oktober 1976. IEEE Computer Society Press.
- [Gra04] Jim Gray. The next database revolution. In *ACM SIGMOD International Conference on Management of Data (SIGMOD 2004)*, Seiten 1–4, New York, NY, USA, Juni 2004. ACM.
- [Här05] Theo Härder. DBMS Architecture - Still an Open Problem. In Gottfried Vossen, Frank Leymann, Peter C. Lockemann und Wolffried Stucky, Hrsg., *11. GI-Fachtagung für Datenbanksysteme in Business, Technologie und Web (BTW 2005)*, Jgg. 65 of *LNI*, Seiten 2–28. GI, März 2005.
- [IDMW08] Florian Irmert, Michael Daum und Klaus Meyer-Wegener. A New Approach to Modular Database Systems. In *Software Engineering for Tailor-made Data Management (SETMDM 2008)*, Seiten 41–45, März 2008.
- [Käs07] Christian Kästner. Aspect-Oriented Refactoring of Berkeley DB. Diplomarbeit, Otto-von-Guericke-Universität Magdeburg, School of Computer Science, Department of Technical and Business Information Systems, Februar 2007.
- [NTNH03] Dag Nyström, Aleksandra Tesanovic, Christer Norström und Jörgen Hansson. The COMET Database Management System. MRTC report ISSN 1404-3041 ISRN MDH-MRTC-98/2003-1-SE, Mälardalen Real-Time Research Centre, Mälardalen University, April 2003.

- [PLKR07] Mario Pukall, Thomas Leich, Martin Kuhlemann und Marko Rosenmueller. Highly configurable transaction management for embedded systems. In *6th workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS 2007)*, Seite 8, New York, NY, USA, März 2007. ACM.
- [Sci07] Edward Sciore. SimpleDB: A Simple Java-Based Multiuser System for Teaching Database Internals. In *38th ACM Technical Symposium on Computer Science Education (SIGCSE 2007)*, Seiten 561–565, New York, NY, USA, März 2007. ACM.
- [SF89] M. E. Segal und O. Frieder. Dynamic program updating: a software maintenance technique for minimizing software downtime. *Journal of Software Maintenance: Research and Practice*, 1(1):59–79, September 1989.
- [TSH04] Aleksandra Tešanović, Ke Sheng und Jörgen Hansson. Application-Tailored Database Systems: A Case of Aspects in an Embedded Database. In *8th International Database Engineering and Applications Symposium (IDEAS 2004)*, Seiten 291–301, Washington, DC, USA, Juli 2004. IEEE Computer Society.