ETH zürich

ETH Library

XRS: ein erweitertes relationales Datenbanksystem zur Verwaltung von technischen Objekten und Versionen

Report

Author(s):
Meier, Andreas; Durrer, K.; Heiser, Gernot; Petry, Erwin; Wälchli, A.; Wälchli, A.; Zehnder, Carl August

Publication date:
1987

Permanent link:
https://doi.org/10.3929/ethz-a-000814178

Rights / license:
In Copyright - Non-Commercial Use Permitted

Originally published in:
ETH Eidgenössische Technische Hochschule Zürich, Institut für Informatik 76

This page was generated automatically upon download from the ETH Zurich Research Collection.
For more information, please consult the Terms of use.

RC.87.16

ETH
Eidgenössische Technische Hochschule
Zürich
Institut für Informatik

A. Meier, K. Durrer, G. Heiser
E. Petry, A. Wälchli, C.A. Zehnder

XRS: Ein erweitertes relationales Datenbanksystem zur Verwaltung von technischen Objekten und Versionen

E. Petry

Konzept und Realisierung einer Versionenverwaltung in einem relationalen Datenbankkern

Eidg. Techn. Hochschule Zürich
Informatikbibliothek
ETH-Zentrum
CH-8092 Zürich
76
87.59

April 1987

Adresse der Autoren:

Institut für Informatik, ETH Zürich
ETH-Zentrum
CH-8092 Zürich / Schweiz

© 1987 Institut für Informatik, ETH Zürich

Vorbemerkungen:

Der vorliegende Bericht enthält zwei selbständige Beiträge zum Projekt XRS, woran die Forschungsgruppe von Dr. Andreas Meier seit 1985 am Institut für Informatik der ETH Zürich arbeitet. Ziel von XRS ist es, einen Prototypen eines Nicht-Standard-Datenbanksystems zu konzipieren und zu implementieren. Dabei werden am klassischen Relationenmodell gezielt Erweiterungen vorgenommen, weshalb das System den Namen XRS für eXtended Relational System trägt.

Die Berichte zeigen den Stand des Projekts XRS im Frühjahr 1987. Im ersten Beitrag werden das Gesamtprojekt sowie seine Realisierung besprochen. Der zweite Aufsatz konzentriert sich auf die Beschreibung einiger Erweiterungen des Datenmodells zur Verbesserung der Objekt- und Versionenverwaltung.

Beide Beiträge sind in sich geschlossen und unabhängig voneinander lesbar. Daraus ergeben sich einige Überschneidungen.

Schlüsselwörter:

erweitertes Relationenmodell, objektorientiertes Datenmodell, technische Datenbank, Datenbankkern, Surrogat, Effizienzsteigerung, technisch-geometrisches Objekt, Version, Versionengraph, Differenzverfahren.

Inhaltsverzeichnis:

A. Meier, K. Durrer, G. Heiser, E. Petry, A. Wälchli, C.A. Zehnder

XRS: Ein erweitertes relationales Datenbanksystem zur Verwaltung von technischen Objekten und Versionen

Zusammenfassung 7

1. Einsatz von Datenbanksystemen in Technik und Wissenschaft 8

2. Erweiterung des relationalen Datenmodells 10
   2.1 Surrogatkonzept 10
   2.2 Erweiterung des Relationenbegriffs 10
      2.2.1 K-Relationen 10
      2.2.2 H-Relationen 11
      2.2.3 M-Relationen 13
   2.3 Versionen 14

3. Architektur und Implementierung des Datenbankkerns 15
   3.1 Überblick über die Systemarchitektur 15
   3.2 Aufbau des Surrogates 16
   3.3 Zugriffsorganisation 17
   3.4 Physische Seitenauslegung für K- und H-Tupel 18
   3.5 Gitterdatei für M-Tupel 22
   3.6 Metadatenbank als H-Relation 22
   3.7 Prozedurale Schnittstelle mit erweitertem Cursorkonzept 23

4. Projektstand und Ausblick 26

Literatur 27

Anhang: Syntax und Semantik der XDS-Schnittstelle 29

E. Petry

Konzept und Realisierung einer Versionenverwaltung in einem relationalen Datenbankkern

Zusammenfassung 33

1. Einleitung 34

2. Ein Modell zur Objektmodellierung und Versionenverwaltung 34

3. Unterstützung von Versionenverwaltung im XRS-Modell 37
   3.1 Ein Surrogatkonzept 37
   3.2 Erweiterte Attributtypen 38
   3.3 Referenzielle Integrität für hierarchische Beziehungen 38
   3.4 Das XRS-Modell unter Berücksichtigung von Effizienzüberlegungen 41
   3.5 Graphen auf Versionen 42

4. Die Implementierung des XRS-Datenbankkerns 44
   4.1 Der Systemaufbau 44
   4.2 Die XDS-Schnittstelle 45
   4.3 Implementierung der Graphen und Differenzen 48

5. Schlussbemerkungen 48

Literaturverzeichnis 50

Anhang: Die XDS-Schnittstelle 52

XRS: Ein erweitertes relationales Datenbanksystem zur Verwaltung von technischen Objekten und Versionen

A. Meier, K. Durrer, G. Heiser, E. Petry, A. Wälchli, C.A. Zehnder

Institut für Informatik
ETH Zürich

Zusammenfassung:
Das erweiterte relationale Datenbanksystem XRS (eXtended Relational System) basiert auf einem Surrogatkonzept und gestattet die effiziente Verwaltung technisch-geometrischer Objekte. "Hierarchische Relationen" erlauben das Modellieren strukturierter Objekte und das systemkontrollierte Identifizieren von Objekten und Teilobjekten. "Mehrdimensionale Relationen" ermöglichen die Beschreibung geometrischer Objekte mit gleichzeitiger Unterstützung von Bereich- und Nachbarschaftsfragen; dieser Relationentyp wird mit Hilfe von Gitterdateien (grid files) realisiert. Für beide neuen Typen von Relationen sind nach wie vor Operationen analog zum klassischen Relationenmodell verfügbar, womit die Einfachheit relationaler Benutzerschnittstellen garantiert bleibt. Neben hierarchischen und mehrdimensionalen Relationen erlaubt XRS, mehrere Tupel als verschiedene Versionen ein und desselben Objekts zu verwalten.

Schlüsselwörter:
Erweitertes Relationenmodell, Technische Datenbank, Surrogat, technisch-geometrisches Objekt, Version.

Summary:
The eXtended Relational database System XRS is based on a surrogate concept in order t allow efficient handling of technical/geometrical objects. "Hierarchical relations" suppon adequate modeling of structured objects and allow to identify objects and parts of objects by system generated surrogates. "Multi-dimensional relations" are used to define geometric objects and to perform range and proximity queries; their implementation is based on grid files. It is important to note that the new types of relations support operations analogous to the classical relational model. Thus, ease of use and simplicity of relational interfaces are maintained. Besides providing hierarchical and multi-dimensional relations, XRS allows to view several tuples of a relation as different versions of the same object.

Keywords:
Extended Relational Model, Engineering Database, Surrogate, technical/geometrical Object, Version.

Computing Reviews Classification: H.2, J.6

Diese Arbeit wird vom Schweizerischen Nationalfonds im Projekt Nr. 2.734-0.85 unterstützt.

1. Einsatz von Datenbanksystemen in Technik und Wissenschaft

Bei vielen Computeranwendungen in Technik und Wissenschaft, etwa bei der rechnergestützten Konstruktion von Maschinenteilen, dem Entwurf integrierter Schaltungen oder bei geographischen Informationssystemen fallen grosse Mengen von strukturierten Daten an, welche längerfristig gespeichert werden müssen. Die Verwaltung solcher Datenbestände in einem konventionellen Dateisystem bringt erhebliche Probleme, insbesondere wenn mehrere Personen am selben Projekt arbeiten. Meist legt jeder Benutzer eigene Dateien an, was unter Umständen zu hoher Redundanz und damit verbundenen Konsistenzproblemen führt. Auch erschweren unterschiedliche Formate den Austausch von Daten zwischen mehreren am gleichen Projekt arbeitenden Personen.

Ähnliche Schwierigkeiten stellten sich Ende der 60er Jahre im administrativen Bereich und führten zu organisierten Datensammlungen, genannt Datenbanken. Im Gegensatz zu einem Dateisystem ist ein Datenbanksystem auf hohe *Datenunabhängigkeit* ausgerichtet, d.h. auf strikte Trennung der Datenhaltung von Anwendungsprogrammen. Somit wird verhindert, dass sich jeder Anwender mit dem Aufbau und der Struktur der Daten befassen muss. Die einheitliche Speicherung erleichtert die gemeinsame Benutzung der Daten. Weiter unterstützt ein Datenbanksystem *Datenintegrität*, da es aufgrund einer vollständigen Datenbeschreibung die Auswirkungen von Operationen zentral (im Fall einer verteilten Datenbank auch dezentral) kontrolliert. Neben dieser Integritätskontrolle sind Massnahmen zur Protokollierung und zum Wiederanlauf sowie zur Synchronisation konkurrierender Programme vorgesehen, womit ein *Mehrbenutzerbetrieb* ermöglicht wird.

Diese und andere Vorteile führten zu intensivem Gebrauch von Datenbanken vor allem im Banken-, Versicherungs- und Verwaltungsbereich. Im Gegensatz dazu ist die Verwendung von Datenbanken in Technik und Wissenschaft noch nicht so verbreitet [Blaser/Pistor 1985]. Dies liegt daran, dass heute übliche Datenbanksysteme für die Bedürfnisse administrativer Anwender entwickelt wurden, die sich von denen der Ingenieure unterscheiden: Im technisch-wissenschaftlichen Bereich lassen sich die Daten im allgemeinen nur schwer mit bestehenden Datenmodellen beschreiben, da sie eine kompliziertere, namentlich eine räumlich-geometrische Struktur aufweisen. Auch das Identifizieren aller Datensätze eines technischen Objektes ist umständlicher als bei administrativen Daten; insbesondere müssen die grafischen Möglichkeiten eines interaktiven Arbeitsplatzes mitberücksichtigt werden. Häufig bestehen mehrere Ausprägungen ein und desselben technischen Objektes, weshalb ein Datenbanksystem eine Versionenkontrolle anbieten sollte. Strukturierte Objekte, Versionen und ein verändertes Zugriffsverhalten (z.B. raumbezogene Abfragen) erfordern neue Speicher- und Zugriffskonzepte. Die Datenbankforschung hat solche Forderungen erkannt und sucht nach neuen Konzepten [Dittrich et al. 1985].

Zur Erweiterung bestehender Datenbanksysteme kann das *NF²-Modell* [Schek/Scholl 1986] dienen, bei welchem Attribute relationenwertige Ausdrücke aufweisen dürfen. Das Darmstädter Datenbanksystem DASDBS [Deppisch et al. 1985] sowie das Advanced Information Management (AIM) Projekt in Heidelberg [Lum et al. 1985] sind Beispiele dieser Entwicklung. *Rekursive Datenmodelle* [Lamersdorf/Schmidt 1983] erlauben das Beschreiben von strukturierten Datenobjekten durch rekursiv definierte Datentypen, wobei Konsistenzbedingungen aus unterschiedlichen Anwendungsbereichen miteinbezogen werden können. Das *Surrogatmodell* [Meier/Lorie 1983a] basiert auf dem klassischen Relationenmodell, führt aber zur Beschreibung von komplexen oder molekularen Objekten

ein spezielles Attribut Surrogat ein [Lorie et al. 1985].

In unserer Arbeit verwenden wir das Surrogatmodell, wobei neben strukturierten Objekten auch räumliche Objekte mit mehrdimensionalen Schlüsseln effizient bearbeitet werden können; zusätzlich bieten wir eine Versionenkontrolle an. Abschnitt 2 erläutert die Konzepte des erweiterten relationalen Datenbanksystems XRS (eXtended Relational System) zur Beschreibung technischer Objekte und Versionen. Abschnitt 3 beschreibt Systemarchitektur und Implementierung des Datenbankkerns von XRS, Abschnitt 4 zeigt Projektstand und geplante Entwicklungen.

2. Erweiterung des relationalen Datenmodells

2.1 Surrogatkonzept

Im klassischen Relationenmodell werden einzelne Tupel einer Relation durch Merkmalswerte identifiziert; der Benutzer deklariert dazu ein spezielles Attribut (oder eine Attributkombination) als Identifikationsschlüssel. Die Benutzerverantwortung für den Identifikationschlüssel führt zu erheblichen praktischen Schwierigkeiten. Beispielsweise können wir vom Benutzer einer grafischen Schnittstelle nicht erwarten, dass er jedem Teil eines technischen Objekts eine eindeutige Identifikation zuordnet. Beim rechnergestützten Konstruieren z.B. zeigt der Benutzer mit Lichtgriffel oder Maus auf ein Bild eines Objekts und erwartet, dass das System grafische Primitiven oder Segmente identifiziert und automatisch den entsprechenden Objekten oder Teilen in der Datenbank zuordnet.

Ein weiteres Problem ergibt sich aus der Veränderlichkeit der Datenbestände etwa beim rechnergestützten Konstruieren. Trotz weitsichtiger Planung ändert sich früher oder später praktisch jedes betriebliche Identifikationssystem. Benutzerdefinierte Identifikationsmechanismen machen es dann schwierig, wenn nicht gar unmöglich, zwei verschiedene Datenbestände auch nur innerhalb einer Unternehmung ohne grösseren Aufwand zu vereinen; meistens wird das Identifikationssystem dann unbrauchbar.

Hall, Owlett und Todd [Hall et al. 1976] erkannten die geschilderten Probleme frühzeitig und führten sie darauf zurück, dass der Benutzer versucht ist, einem Schlüsselwert eine Bedeutung zu geben. Zur Lösung schlugen sie die Einführung von Surrogaten vor: *Surrogate* sind invariante, vom System zugeteilte Identifikationsschlüssel der Datensätze in einer Datenbank; sie können an verschiedenen Stellen innerhalb der Datenbank zur Definition von *Beziehungen* benutzt werden. Im Gegensatz dazu sind benutzervergebene Schlüssel unter der Kontrolle des Anwenders und ihre Invarianz wird nicht garantiert.

Die Einführung von Surrogaten eignet sich besonders als Basis von Datenbanken für den Ingenieur [Meier 1986a]. Die meisten der erwähnten Mängel herkömmlicher Identifikationssysteme in Datenbanken lassen sich eliminieren, falls man an die Surrogatwerte folgende Forderungen stellt:

- Jeder Surrogatwert ist innerhalb des Datenbanksystems *eindeutig*.
- Das Surrogat wird *vom System ohne Benutzereinfluss vergeben*. Die Surrogatwerte können vom Benutzer verwendet, aber nicht verändert werden.
- Ein einmal vergebener Surrogatwert wird *nicht wiedervergeben*, auch dann nicht, wenn das zugehörige Objekt nicht mehr existiert.

Das Surrogat bildet ein mächtiges Instrument für den Aufbau technischer Datenbanken. Es erlaubt, strukturierte Objekte durch systemvergebene Werte anstelle von Benutzerschlüsseln zu identifizieren, gleichzeitig kann es auch vom System zur effizienten Organisation der Daten ausgenutzt werden.

2.2 Erweiterung des Relationenbegriffs

2.2.1 K-Relationen

Im klassischen Relationenmodell ist eine Relation definiert als Teilmenge des kartesischen Produkts von Attributwertebereichen. Ein Attribut oder eine Attributkombination dient

darin direkt als Identifikationsschlüssel. Dieser Relationenbegriff lebt in unserem Modell in leicht modifizierter Form als klassische Relation oder K-Relation weiter. Wir entlasten jedoch den Benutzer von der Verantwortung für den Schlüssel und stellen statt dessen das Surrogat zur Verfügung.

Definition: Eine *K-Relation* R der Dimension n ist eine Teilmenge des kartesischen Produkts aus Surrogat S und n Wertebereichen $D_1,...,D_n$ mit S als Identifikationsschlüssel, d.h.

$$R \subset S \times D_1 \times ... \times D_n.$$

Die K-Relation R ist somit eine Menge von Tupeln $(s, d_1, ..., d_n)$, die *K-Tupel* genannt werden. Die Wahl der Wertebereiche D_i ist im klassischen Relationenmodell durch die Forderung der Ersten Normalform beschränkt: Attributwertebereiche dürfen keine Struktur aufweisen. In XRS lassen wir darüber hinaus gewisse strukturierte Typen wie Vektoren zu. Darauf wird im letzten Kapitel noch eingegangen werden.

Mit dem Surrogat ist ein direkter Zugriff auf ein Tupel möglich. Neben dem Surrogat können in gewohnter Weise als Zugriffshilfen "Benutzerschlüssel" deklariert werden. Ein *Benutzerschlüssel* ist ein beliebiges Attribut oder eine Attributkombination. Der Zugriff über Benutzerschlüssel wird vom System besonders unterstützt, während der Zugriff über Nichtschlüsselattribute ein Durchsuchen der gesamten Datenbank erfordert.

Beispiel: Eine K-Relation könnte etwa wie folgt definiert sein:

```
RELATION Object
  ATTRIBUTE
    objectId:SURROGATE
    kind:    ObjectKind
    type:    String20
    ...
  KEY
    kind                        /* Benutzerschlüssel */
END Object
```

Als Standardoperationen auf K-Relationen setzen wir die klassische Relationenalgebra mit Vereinigung, Differenz, kartesischem Produkt, Projektion und Selektion voraus; zusätzlich sei der Verbundoperator speziell erwähnt, wobei wir uns bei all diesen Operatoren an die üblichen Definitionen halten (vergl. z.B. [Zehnder 1987]).

2.2.2 H-Relationen

Das wesentliche Konzept zur Modellierung technischer Objekte in unserer Datenbank ist die *hierarchische Relation* oder *H-Relation*

Unter einer H-Relation verstehen wir ein hierarchisches (genauer: baumartiges) Konstrukt von K-Relationen. Sie besitzt eine ausgezeichnete *Wurzelrelation*, der "abhängige Relationen" zugeordnet sind. Diese Zuordnung geschieht dadurch, dass ein "abhängiges Tupel" das Surrogat des direkt übergeordneten Tupels als speziellen Attributwert enthält.

Definition: Eine von einer Relation R_i *abhängige Relation* R_j ist Teilmenge des kartesischen

Produkts aus Surrogat S_j, Fremdsurrogat S_i und n Wertebereichen $D_1,...,D_n$, d.h.

$$R_j \subset S_j \times S_i \times D_1 \times ... \times D_n.$$

Die Relation R_i ist entweder die Wurzelrelation oder selbst eine abhängige Relation. S_i ist das Surrogat von R_i; es wird auch als *H-Referenz* bezeichnet. Die Tupel $(s_j, s_i, d_1, ..., d_n)$ heissen *abhängige Tupel.*

Definition: Eine *H-Relation* besteht aus einer K-Relation als Wurzelrelation und aus einer Hierarchie abhängiger Relationen.

Ein *H-Tupel* ist ein Tupel aus der Wurzelrelation mit all seinen abhängigen Tupeln.

Selbstverständlich sind für Nicht-Surrogat-Attribute auch beliebige Wertebereiche, und damit auch Fremdsurrogate zugelassen. Solche allgemeinen Referenzen führen zu netzwerkartigen Konstruktionen. Diese werden von unserem System jedoch nicht auf Konsistenz überprüft. Im Gegensatz dazu steht die H-Referenz unter Systemkontrolle und muss deshalb besonders gekennzeichnet werden. Dies kann etwa durch die Konstrukte PART-OF oder IS-A geschehen [Meier 1986a].

Beispiel: Klassische Stückliste:

```
RELATION Face
  ATTRIBUTE
     faceId:  SURROGATE
     objectId:PART-OF (Object)  /* H-Referenz */
     color:   (red,green,blue)
     ...
  END Face
```

Beispiel: Generalisierungshierarchie:

```
RELATION Cylinder
  ATTRIBUTE
     cylId:   SURROGATE
     primId:  IS-A (Primitive)  /* H-Referenz */
     radius:  Real
     height:  Real
     ...
  END Cylinder
```

Da abhängige Relationen und Wurzelrelationen spezielle K-Relationen sind, sind auf ihnen die üblichen Operationen anwendbar. Basierend auf dem Surrogatmodell bieten wir darüber hinaus mächtige Operationen auf H-Relationen an, da die hierarchischen Beziehungen dem Datenbanksystem bekannt sind und deshalb zur effizienten Verwaltung der Daten ausgenutzt werden können. Als wichtigste Operationen auf H-Relationen gelten der ***implizite hierarchische Verbund*** [Meier/Lorie 1983b] sowie mengenorientierte Operationen zur Versionenkontrolle (siehe Abschnitt 3.3).

2.2.3 M-Relationen

In rechnergestützten Entwurfssystemen, geographischen Informationssystemen und anderen zwei- oder dreidimensionalen Geometriesystemen sind geometrische Nachbarschaftsbeziehungen zwischen den Daten von grosser Bedeutung. Um diesbezügliche Anfragen effizient unterstützen zu können, führen wir die *mehrdimensionale Relation* oder M-Relation ein. Sie ist eine spezielle K-Relation, die sich durch die Existenz eines besonderen Schlüssels, bestehend aus mehreren gleichberechtigten Attributen, auszeichnet.

Definition: Eine *M-Relation* R der Dimension m+n ist Teilmenge des kartesischen Produkts aus Surrogat, m Schlüsselbereichen $K_1,...,K_m$ (m>1) und n Wertebereichen $D_1,...,D_n$, d.h.

$$R \subset S \times K_1 \times ... \times K_m \times D_1 \times ... \times D_n.$$

Die zu den Schlüsselbereichen $K_1,...,K_m$ gehörende Attributkombination heisst *mehrdimensionaler Schlüssel*. Alle Schlüsselbereiche K_i sind gleichberechtigt, d.h. auf alle sind dieselben Operationen anwendbar und die physische Datenorganisation bevorzugt keinen Schlüsselbereich gegenüber einem anderen. Die Tupel $(s, k_1, ..., k_m, d_1, ..., d_n)$ der M-Relation heissen *M-Tupel*

Beispiel:

```
RELATION Point
  ATTRIBUTE
    pointId: SURROGATE
    x:       REAL
    y:       REAL
    z:       REAL
    type:    String10
    ...
  M-KEY                              /* mehrdimensionaler */
    x,y,z                            /* Schlüssel         */
END Point
```

Auch auf mehrdimensionalen Relationen sind natürlich die Operationen der Relationenalgebra anwendbar. Ihre Bedeutung liegt aber vor allem in der Unterstützung von *Punkt- und Bereichsanfragen*. Wenn wir im Moment die Nichtschlüsselattribute der entsprechenden Wertebereiche $D_1,...,D_n$ sowie das Surrogat vernachlässigen, so können wir sämtliche Tupel der M-Relation R als Punkte eines m-dimensionalen Parameterraumes auffassen. Die folgenden Anfragetypen [Meier 1986b] sind möglich und werden speziell bei M-Relationen unterstützt:

Punktfrage: Bei der Angabe von m Schlüsselwerten soll das entsprechende Tupel gefunden werden, falls es existiert.

Teilpunktfrage: Anstelle von m Schlüsseln gibt man eine Schlüsselkombination mit weniger als m Schlüsseln vor und interessiert sich für alle Tupel, die dieser Schlüsselkombination genügen.

Bereichfrage: Für jeden der m Schlüssel spezifiziert man einen Bereich. Sämtliche Tupel erfüllen die Anfrage, deren Schlüssel in den jeweiligen Bereichen liegen.

Teilbereichfrage: Analog der Bereichfrage, wobei weniger als m Bereiche spezifiziert werden.

Weitere Operationen auf M-Relationen sind möglich. So können etwa Nachbarschaftsanfragen in die Relationenalgebra eingebettet werden. Mit Hilfe eines Abstandskriteriums (z.B. Euklidsche Metrik bei geometrischen Daten) und eines Referenzpunktes lassen sich beispielsweise sämtliche Tupel selektieren, die höchstens einen maximalen Abstand vom Referenzwert aufweisen.

2.3 Versionen

Zur effizienten Verwaltung logisch zusammengehörender Objekte derselben Art bietet unser Datenbanksystem ein zweistufiges Versionenkonzept mit Versionen und Versionenmengen an. Dabei werden alle Tupel, die eine Version eines Objekts beschreiben, in derselben K-, H- oder M-Relation abgelegt. Das gilt auch für die zweite Stufe, in der mehrere Versionen zu einer *Versionenmenge* zusammengefasst werden können. Die Tupel dieser Versionenmenge werden dann als Repräsentanten verschiedener *Versionen* ein und desselben Objekts betrachtet. Jede Version gehört genau einer Versionenmenge an, weshalb die Versionenmengen eine vollständige Partition einer Relation darstellen.

Versionenmengen bilden eine zusätzliche Navigationseinheit für Datenbankabfragen. Sie können ähnlich wie ganze Relationen manipuliert werden. Daneben bleibt die herkömmliche relationale Sicht auf die Einzelobjekte (Versionen) erhalten, d.h. bei einer Abfrage kann die Tatsache, dass Tupel zu Versionenmengen zusammengefasst sind, ignoriert werden.

Die für Versionen verfügbaren Operationen sind Definition und Manipulation einzelner Versionen. Die Definition einer Version kann dabei entweder durch Neueinfügen eines Tupels geschehen oder aber durch Kopieren einer bestehenden Version. (In diesem Fall ist offen, ob das System eine wirkliche Kopie des physischen Objekts anlegt, oder ob es durch ein Differenzverfahren einen Verweis auf das ursprüngliche Objekt sowie eventuelle Änderungen speichert.) Die Manipulationsoperationen erlauben den Zugriff auf Versionen oder Versionenmengen sowie das Ändern einzelner Versionen. Weiter können Versionen oder ganze Versionenmengen gelöscht werden. Es wird bewusst darauf verzichtet, Operationen anzubieten, die Versionen zwischen verschiedenen Versionenmengen bewegen, da Versionen aus verschiedenen Versionenmengen Repräsentanten verschiedener Objekte darstellen. Eine Version ist an ihre Versionenmenge gebunden, ebenso wie ein Tupel an seine Relation gebunden ist.

Der Benutzer wird nicht gezwungen, sich von vorneherein festzulegen, ob er die Möglichkeit der Versionenbildung in Anspruch nehmen will. Es ist in einer Relation, die bisher nicht versionenbehaftet war, jederzeit möglich, Tupel als Teil einer Versionenmenge zu betrachten und damit nachträglich von der Versionenverwaltung Gebrauch zu machen.

Bei H-Relationen ist Versionenbildung nur auf der vollständigen Hierarchie möglich. Es können also nur Versionen vollständiger Objekte existieren, nicht aber von Teilen von Objekten.

3. Architektur und Implementierung des Datenbankkerns von XRS

3.1 Überblick über die Systemarchitektur

In Abb. 1 zeigen wir die Systemarchitektur des Datenbankkerns von XRS, der sich im wesentlichen aus vier Schichten zusammensetzt (vgl. Schichtenmodell nach [Härder/Reuter 1985]). Die unterste Schicht stellt eine portable Systemschnittstelle dar, worauf als nächste Schicht ein virtuelles Speicherkonzept aufbaut. Die dritte Schicht enthält die Kernkomponenten wie Verwaltung des Datenbankschemas, Surrogatvergabe und -kontrolle, Objektverwaltung, Zugriffspfade und mehrdimensionale Zugriffsunterstützung. Die vierte Schicht umfasst eine prozedurale Schnittstelle mit Funktionen zur Datenbankmanipulation und -abfrage. Darin integriert sind administrative Funktionen, sowie die Datenbankdefinition und Versionenkontrolle.

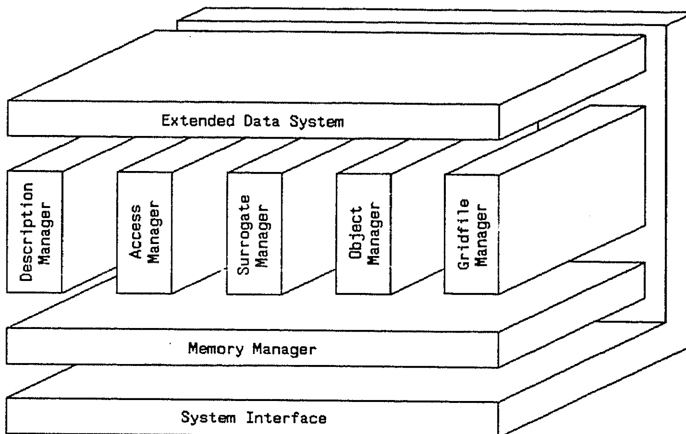


Abb. 1: Aufbau des Datenbankkerns von XRS

Im folgenden werden die einzelnen Komponenten des Datenbankkerns von XRS charakterisiert:

Extended Data System (XDS)

Diese Schicht umfasst eine prozedurale Schnittstelle, welche Funktionen zur Datenbankeröffnung und -schliessung, zur Abfrage und Manipulation von Objekten, Versionen oder Schemainformationen sowie zur Transaktionssteuerung anbietet. Die Prozeduren arbeiten zum Teil satzorientiert (Abfrage-, Einfüge- und Ersetzoperationen), zum Teil mengenorientiert (Lösch- und Kopieroperationen), wobei ein Cursorkonzept die Abfrage- und Manipulationsoperationen unterstützt. Als wesentliche Aufgabe koordiniert die XDS-Schnittstelle alle Aufrufe der darunterliegenden Schicht, welche eine Vielfalt von Aufgaben parallel bearbeitet.

Description Manager

Diese Komponente verwaltet eine hauptspeicherresidente Beschreibung der aktuellen

Datenbank, d.h. das Datenbankschema.

Surrogate Manager
Hier werden unter Berücksichtigung der Eindeutigkeitsforderungen neue Surrogatwerte vergeben und auf physische Adressen abgebildet.

Access Manager
Die Zugriffspfade von Benutzerschlüsseln auf Surrogate werden mit B*-Bäumen verwaltet und nachgeführt.

Object Manager
Diese Komponente verwaltet K- und H-Tupel sowie deren Versionen, insbesondere auch die Beschreibungsdaten.

Gridfile Manager
Diese Kernkomponente verwaltet die M-Tupel und ermöglicht den Zugriff über mehrdimensionale Schlüssel. Sie basiert auf der Gitterdatei [Nievergelt et al. 1985].

Memory Manager
Diese Schicht verwaltet eine variabel festlegbare Anzahl Seiten im Hauptspeicher unter Verwendung eines einfachen, virtuellen Konzepts.

System Interface
Die tiefste Schicht des Datenbankkerns von XRS stellt eine portable Standardschnittstelle zur Verfügung. Sie enthält zudem die Definition aller wichtigen Datenstrukturen von XRS sowie die Werte der globalen Datenbankparameter.

Querzugriffe zwischen Komponenten der dritten Schicht werden nach Möglichkeit vermieden, die Kommunikation wird über die vierte Schicht vollzogen. Diese Modularisierung wird von der verwendeten Programmiersprache Modula-2 [Wirth 1985] weitgehend unterstützt. Die standardisierte Schnittstelle OSSI [Biagioni et al. 1986] als Teil der Systemschnittstelle garantiert Geräteunabhängigkeit.

3.2 Aufbau des Surrogats

Damit das Surrogat als Identifikationsschlüssel sowohl abhängige Tupel einer H-Relation als auch beliebige Tupel einer Versionenmenge eindeutig bezeichnen kann, wird es wie folgt festgelegt:

$$\text{Surrogat} = [R\#, S\#, V\#, N\#]$$

Die Bedeutungen der einzelnen Teile des Surrogats sind:

R#: Nummer der Relation, zu der das Tupel gehört (Relation number).

S#: Identifikation der Versionenmenge innerhalb einer Relation (version Set number).

V#: Eindeutige Versionennummer (Version number) innerhalb einer Versionenmenge.

N#: Laufnummer (sequence Number). Da in einer abhängigen Relation mehrere Tupel zur gleichen Versionenmenge und Version gehören können, ist dieser vierte Surrogatteil zur eindeutigen Identifizierung eines abhängigen Tupels notwendig.

Aus dem Surrogat ergibt sich im Sinne eines zusammengesetzten, hierarchischen Schlüssels die Identifikation der Relation innerhalb der Datenbank, der Versionenmenge innerhalb der Relation und der Version innerhalb der Versionenmenge des zugehörigen Tupels.

Die Versionenbildung ist mit keinerlei Mehraufwand für die physische Verwaltung der Tupel verbunden, da sie durch die Struktur des Surrogats getragen wird. Jede vom Benutzer nicht als Version eines Objekts bezeichnete Ausprägung bildet eine 1-elementige Versionenmenge. Dadurch können dem Benutzer Operationen zur Verfügung gestellt werden, mit denen er jederzeit, auch nachträglich, Ausprägungen als Versionen eines Objekts betrachten kann.

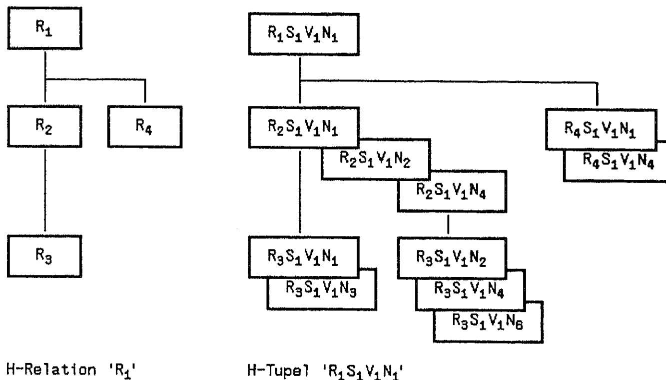


Abb. 2: H-Relation mit Ausprägung zur Veranschaulichung des Surrogats

Die Bedeutung der verschiedenen Teile des Surrogats wird in Abb. 2 illustriert. Wichtig dabei ist, dass sich der Versionenbegriff nur auf ein vollständiges Objekt beziehen kann; aus diesem Grund sind innerhalb eines H-Tupels die Surrogatteile S# und V# immer gleich.

3.3 Zugriffsorganisation

Die Zugriffsorganisation dient dazu, aufgrund einer Anfrage diejenigen Datensätze zu finden, welche die vom Benutzer definierten Selektionskriterien erfüllen. Im Datenbankkern von XRS besteht eine dreistufige Zugriffsorganisation, nämlich über Benutzerschlüssel, Surrogate sowie mehrdimensionale Schlüssel.

Wie aus Abb. 3 hervorgeht, werden im Datenbankkern von XRS zwei unterschiedliche Dateiorganisationen verwendet: eine punktbezogene und eine raumbezogene. K- und H-Tupel werden auf eigenen Datenseiten abgelegt (siehe Abschnitt 3.4). Zu deren Verwaltung dienen virtuelle Speicheradressen, genannt TID (Tuple IDentifier). Für die Ausprägungen von M-Relationen wird die raumbezogene Gitterdatei verwendet (siehe Abschnitt 3.5). Die Gitterdatei stellt ein Adressberechnungsverfahren dar, d.h. einem vollständig spezifizierten mehrdimensionalen Schlüssel entspricht direkt die physische Adresse eines M-Tupels.

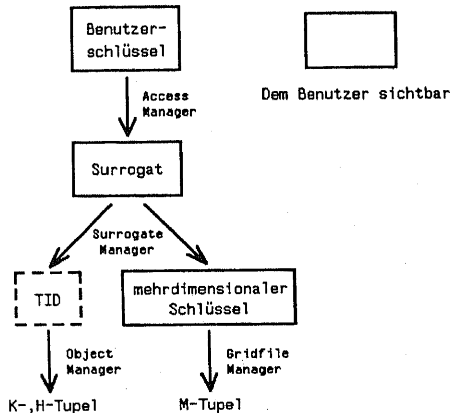


Abb. 3: Dreistufige Zugriffsorganisation über Benutzerschlüssel, Surrogat und mehrdimensionalen Schlüssel

Mit dem Surrogat wird jedes Tupel jeder K-, H- oder M-Relation eindeutig identifiziert. Es steht damit im Zentrum der Zugriffsorganisation. Das Surrogat wird auf TIDs (K- und H-Relationen) bzw. mehrdimensionale Schlüssel (M-Relationen) abgebildet. Diese Abbildung geschieht durch eine Gitterdatei, für die das Surrogat selber wiederum als vierdimensionaler Schlüssel aufgefasst wird. Hierdurch wird ein effizienter Zugriff auf die Menge der zu einem unvollständig spezifizierten Surrogat gehörenden Tupel ermöglicht.

Für den beschleunigten Zugriff auf Daten steht es dem Benutzer frei, neben Surrogat und mehrdimensionalen Schlüsseln Benutzerschlüssel zu definieren. Diese werden über B*-Bäume auf Surrogate abgebildet.

3.4 Physische Seitenauslegung für K- und H-Tupel

Zur Verwaltung von K- oder H-Tupeln auf physischen Datenseiten wird das indirekte TID-Konzept nach [Härder 1978] benutzt, um Änderungen innerhalb einer Seite möglichst lokal behandeln zu können: Die Adresse eines Tupels besteht aus zwei Teilen. Die erste Komponente bezeichnet die Seitennummer, die zweite stellt einen Index in eine Tabelle dar, die sich auf derselben Seite befindet und die relative Position in der Seite enthält.

Dank dem indirekten Adressberechnungskonzept haben Verschiebungen von Daten innerhalb einer Seite keine Änderungen im Zugriffspfad zur Folge, da sich die TIDs nicht ändern.

Um auch Manipulationen an Tupeln effizient zu vollziehen, die Verschiebungen von Daten über Seitengrenzen hinaus zur Folge haben, ist eine möglichst grosse Unabhängigkeit der Seiten anzustreben. Im Datenbankkern von XRS werden deshalb nur die Seiten selbst untereinander verkettet (Einträge an einem festen Platz im Seitenkopf), ansonsten sind keinerlei datenspezifische Angaben vorhanden, die über die Seiten hinweg verweisen.

Die wesentliche Anforderung an die physische Datenorganisation von XRS ist die hierarchische Clusterung von K- und H-Relationen. Dies bedeutet, dass auf unterster Ebene abhängige Tupel eines H-Tupels möglichst zusammen mit den übergeordneten Tupeln gespeichert werden sollen. Zudem sollen alle zur selben Versionenmenge gehörenden Tupel sowie alle Versionenmengen einer Relation zusammen gespeichert werden. Die Clusterung beruht auf der hierarchischen Ordnung "Relation-Versionenmenge-Version", welche durch die Einführung zweier virtueller Relationen SysR (System Relation) und SysV (System Version Set) pro K- oder H-Relation realisiert wird. Die beiden Systemrelationen SysR und SysV sind für den Benutzer nicht sichtbar. Sie beanspruchen auch keinen Speicherplatz, da deren Tupel keinerlei Information enthalten, die nicht ohnehin abgespeichert werden müsste.

Pro K- oder H-Relation gibt es ein Tupel in SysRel, das pro Versionenmenge genau ein abhängiges Tupel aus SysVersionSet besitzt. Dieses wiederum enthält als abhängige Tupel genau die K- oder H-Tupel der benutzerdefinierten Relation. Wird der Baum von abhängigen Tupeln nun in hierarchischer Ordnung (Präordnung) gespeichert, so ergibt sich auf natürliche Weise die erwünschte hierarchische Clusterung der K- und H-Relationen.

Die hierarchische Organisation erlaubt es, für K- und H-Tupel, versionenbehaftet oder nicht, mit ein und derselben Seitenauslegung auszukommen. Wir erläutern diese anhand eines Beispiels.

Abb. 4 stellt eine mögliche Ausprägung einer H-Relation dar, welche aus einer Wurzelrelation A und abhängigen Relationen B und D besteht, wobei B eine weitere abhängige Relation C umfasst.

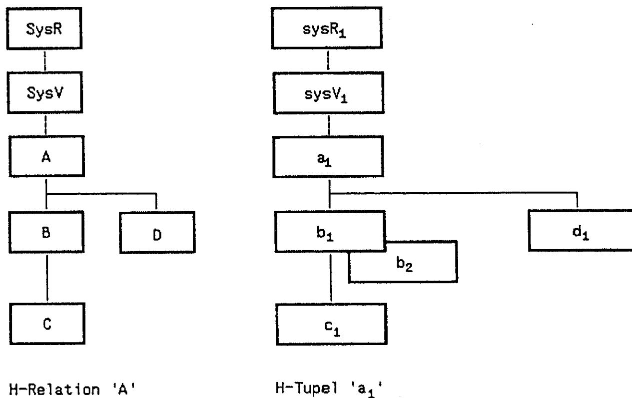
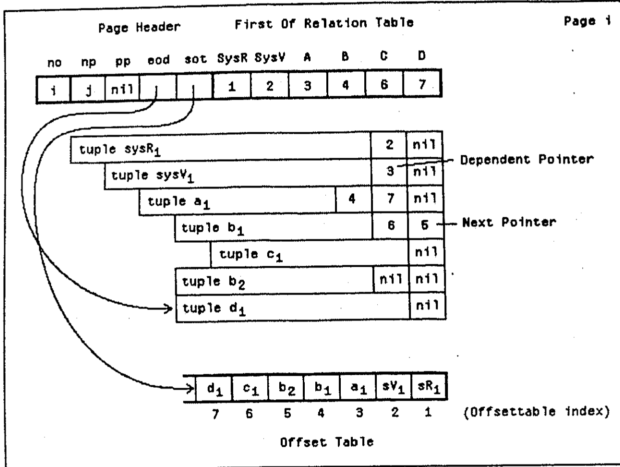


Abb. 4a: H-Relation 'A' mit Ausprägung



Legende:

no: Seitennummer
np: Verweis auf logisch nächste Seite
pp: Verweis auf logisch vorangehende Seite
eod: Ende der Dateneinträge
sot: Beginn der Offsettabelle

SysR: Systemrelation 'Relation'
SysV: Systemrelation 'Version Set'
A: Wurzelrelation
B,..: Abhängige Relationen

Dependent Pointer : Verweis auf erstes Tupel einer abhängigen Relation

Pro abhängige Relation existiert ein Dependent Pointer

Next Pointer : Verweis auf nächstes Tupel innerhalb derselben Relation

'nil'-Werte: Es wird nur auf Tupel verwiesen, die sich auf derselben Seite befinden, d.h. das entsprechende Tupel kann sich evtl. auf einer nachfolgenden Seite befinden (oder es gibt gar kein solches mehr).

Abb. 4b: Auslegung einer physischen Seite für K- und H-Tupel

Die *Seitennummer* (no) wird vom Speicherverwalter vergeben und steht an einem reservierten Ort auf einer Seite. Aus Sicherheitsgründen kann die Seitennummer nach ihrer Vergabe nur noch gelesen werden.

Der *Seitenkopf* (Page Header) schliesst an das für die Seitennummer speziell reservierte Feld an und enthält einen Verweis auf die logisch nächste Seite (next page), auf die logisch vorhergehende Seite (previous page), einen Verweis auf das Ende der Dateneinträge auf dieser Seite (end of data) und einen Zeiger auf den Beginn der Offsettabelle (start of offset table).

Die *FORT*-Tabelle (First Of Relation Table) folgt auf den Seitenkopf. Sie enthält für die beiden Systemrelationen SysR und SysV je einen Verweis auf den ersten Tupeleintrag auf der Seite. In der hierarchischen Reihenfolge A, B, C, D besitzt zudem jede Relation der H-Relation ein FORT-Feld, worin ebenfalls ein Verweis auf das erste Tupel der jeweiligen Relation auf dieser Seite steht; im Falle einer K-Relation existieren natürlich keine abhängigen Relationen. Alle Verweise sind indirekt: In jedem FORT-Feld steht der Index

eines Offsettabellenfeldes.

Die *Offsettabelle* (Offset Table) enthält die relative Seitenposition eines Tupels. Ihre Einführung ermöglicht, dass sich der TID eines Tupels bei einer Verschiebung innerhalb der Seite nicht ändert. Damit sind wie oben erwähnt erheblich weniger Nachführungen des Zugriffspfads von Surrogaten auf physische Adressen nötig.

Tupel, die sich über mehrere Seiten erstrecken, werden streng sequentiell über diese Seiten hinweg abgelegt, bei H-Tupeln unter Erhaltung der hierarchischen Ordnung. Dabei hat das Einfügen und Löschen von Tupeln meistens nur lokale Auswirkungen, solange ein einzufügendes Tupel auf einer Seite noch Platz findet bzw. ein gelöschtes Tupel eine noch genügend gefüllte Seite zurücklässt. Im schlimmsten Fall ist genau eine weitere Seite betroffen. Findet nämlich ein einzufügendes Tupel keinen Platz auf einer Seite, so wird die Seite geteilt; lässt ein gelöschtes Tupel eine Seite ungenügend gefüllt zurück und ist eine Nachbarseite ebenfalls ungenügend gefüllt, so werden diese zwei Seiten zusammengelegt.

Jedes Tupel besitzt neben dem eigentlichen Datensatz einen sogenannten Verweisteil, welcher Zeiger auf andere Datensätze enthält. Zwei Typen von Zeigern werden unterschieden. Jedes K- oder H-Tupel besitzt einen Zeiger auf das nächste Tupel innerhalb derselben Relation (Next Pointer). Im Falle eines H-Tupels existiert pro abhängige Relation ein Zeiger (Dependent Pointer), der auf das jeweils erste abhängige Tupel verweist. Dieser Zeigertyp ist Träger der Strukturinformation eines H-Tupels. Der Verweis ist ein Index eines Offsettabellenfeldes, sofern das zu referenzierende Tupel auf derselben Seite liegt, andernfalls 'nil'.

Nachtrag zum Laufzeitverhalten:

Das Laufzeitverhalten eines Datenbanksystems wird im wesentlichen bestimmt durch die Anzahl Zugriffe auf den Sekundärspeicher, bei Manipulationen an Datenobjekten also durch die Auswirkungen solcher Operationen auf die physischen Seiten. Im Datenbankkern von XRS lassen sich die Operationen auf Seiten durch drei Parameter steuern, die sich während des Betriebs der Datenbank über eine spezielle Prozedur 'Tune' jederzeit ändern lassen. Dabei bewirken Änderungen eines dieser Parameter keine Reorganisation der Datenbank, sondern sind erst bei künftigen Operationen wirksam. Gerade deshalb bieten sie aber eine gute Möglichkeit, umfangreiche Manipulationsarbeiten an K- und H-Tupeln der jeweiligen Objektumgebung individuell anzupassen.

Die *Seitenbelegungsgrenze* gestattet die erstmalige Füllung einer Seite nur bis zu dieser Grenze, ist jedoch bei späterem Einfügen unwirksam. Der Parameter ermöglicht es dem Benutzer, sich häufig in der Grösse ändernde Objekte mit einer niedrigeren Seitenbelegungsgrenze in die Datenbank einzufügen, und so allzu häufiges Teilen und Vereinen von Seiten zu vermeiden.

Eine Seitenbelegung unterhalb der *Seitensterbegrenze* wird durch einen speziellen Vermerk gekennzeichnet. Besitzen zwei Nachbarseiten je einen solchen Vermerk, so werden diese beiden Seiten zusammengelegt. Dieser Parameter kann dazu verwendet werden, Seiten, auf denen viel gelöscht und nachher wieder eingefügt werden soll, vor dem Zusammenlegen zu schützen (indem man die Seitensterbegrenze hinuntersetzt).

Die *Seitenreduziergrenze* gibt an, wo eine zu teilende Seite gespalten wird (falls ein Tupel auf einer bereits vollen Seite eingefügt werden soll). Die betroffene Seite wird dann bis zur Seitenreduziergrenze geleert und der Rest auf die nächste oder eine neu angeforderte Seite verschoben. Mit diesem Parameter kann ein häufiges Einfügen an fast denselbem Ort, wie das bei H-Relationen oft der Fall ist, unterstützt werden (Vermeiden repetitiven Teilens derselben Seite).

3.5 Gitterdatei für M-Tupel

Die Speicherorganisation für M-Tupel unterscheidet sich wesentlich von jener für K- und H-Tupel. Dies ist erforderlich, da sich die auf M-Tupeln angebotenen Operationen mit herkömmlichen Speicherorganisationsformen nicht effizient unterstützen lassen. M-Tupel werden daher in Gitterdateien (grid files) [Nievergelt et al. 1984] abgespeichert. Hierzu verwenden wir die Implementierung der Gitterdatei von [Hinrichs 1985].

Die Gitterdatei ist eine symmetrische, mehrdimensionale, dynamische Datenstruktur auf Sekundärspeicher. Der durch die (gleichberechtigten) Schlüsselbereiche gegebene mehrdimensionale Datenraum wird durch ein orthogonales Gitter unterteilt. Die Schnittpunkte der raumteilenden Hyperebenen mit den Achsen bilden die *Skalen*. Das *Gitterverzeichnis* (Grid-Directory) ordnet jedem durch die Skalen gebildeten Teilraum eine *Datenseite* zu, in welcher die Benutzerdaten gespeichert sind. Da das Gitterverzeichnis im allgemeinen gross ist, wird es selbst als Gitterdatei verwaltet, dessen Gitterverzeichnis (Root-Directory) zusammen mit den zugehörigen Skalen permanent im Hauptspeicher liegt.

Findet ein Datensatz auf der im Gitterverzeichnis vermerkten Datenseite keinen Platz mehr, so muss eine neue Datenseite angefordert und im Gitterverzeichnis eingetragen werden. Dazu kann es notwendig sein, eine neue Hyperebene in das mehrdimensionale Gitter einzubauen. Um die Symmetrie zu gewährleisten, werden neue Hyperebenen zyklisch in den verschiedenen Schlüsselbereichen angelegt. Der Inhalt der vollen Datenseite wird auf zwei Datenseiten (die neu angeforderte und die bisherige) verteilt. Für alle anderen Datenseiten, die ebenfalls von der neuen Hyperebene durchschnitten werden, ist lediglich ein Verweis im Gitterverzeichnis nötig. Wenn die Belegungsquote einer Datenseite niedriger wird als eine bestimmte (wählbare) Schranke, so wird sie mit Nachbarseiten zusammengelegt. Dies gewährleistet eine nahezu optimale Anpassung der Gitterstruktur an die Datenverteilung.

Die Gitterdatei erlaubt effizienten Zugriff auf die mehrdimensionalen Daten; ein einzelner Datensatz kann bei vollständig spezifizierten Schlüsseln stets mit maximal zwei physischen Leseoperationen in den Hauptspeicher geholt werden. Zugriffe mit teilweise spezifizierten Schlüsseln sowie verschiedene Typen von Nachbarschaftsanfragen werden ebenfalls effizient unterstützt, da die Daten unter weitgehender Bewahrung von geometrischer Nachbarschaft abgespeichert werden.

3.6 Metadatenbank als H-Relation

Die Beschreibung des Aufbaus einer Datenbank, das *Datenbankschema*, enthält alle Strukturinformation über die Datenbank, so etwa, welche Relationen die Datenbank enthält, von welchem Typ diese sind, sowie die Beschreibung ihrer Attribute und Schlüssel. Darüber hinaus enthält sie auch Angaben, die im Datenbanksystem intern benötigt und verwaltet werden (wie z.B. die Adresse der ersten Datenseite einer Relation).

Das Datenbankschema muss wie alle Benutzerdaten die gesamte Lebenszeit der Datenbank überdauern und deshalb permanent auf Sekundärspeicher abgelegt werden. Hierzu kann man entweder spezielle Seiten verwenden oder das Datenbankschema selbst als Teil der zu verwaltenden Daten betrachten und in den Relationen des *Metadatenbankschemas* als *Metadatenbank* speichern. Während im ersten Fall zur Verwaltung des Datenbankschemas eigene Software notwendig ist, kann bei einer Metadatenbank das Lesen und Verändern des

Datenbankschemas mit den gleichen Verwaltungsprozeduren wie bei den Benutzerdaten erfolgen. In XRS ist das Datenbankschema als Metadatenbank organisiert.

Das Metadatenbankschema von XRS ist eine H-Relation (Abb. 5). Die Relation MDBS stellt die Wurzel des Metadatenbankschemas dar. Sie dient der physischen Clusterung der Metadatenbank und ersetzt damit die hier nicht vorhandenen Systemrelationen.

Durch die Realisierung der Metadatenbank als H-Relation können einige Konsistenzregeln bereits durch den Objektverwalter erzwungen werden. Beispielsweise können so keine Attributbeschreibungen in die Relation 'Attribute' eingefügt werden, ohne dass die zugehörige Relationenbeschreibung aus 'Relation' bereits existiert. Alle weiteren Konsistenzbedingungen auf der Metadatenbank werden explizit überwacht.

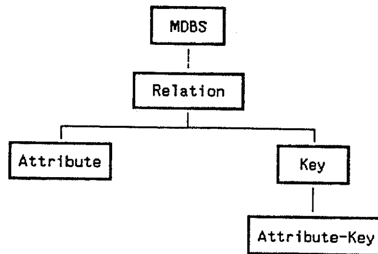


Abb. 5: Das Metadatenbankschema von XRS.

In XRS kann das Datenbankschema jederzeit um neue Relationen erweitert werden. Hingegen werden Schemaänderungen, die eine Reorganisation der Datenbank verlangen, nicht unterstützt.

Da die Metadatenbank bei jeder Operation auf dem Datenbestand benötigt wird, ist ein effizienter Zugriff auf ihren Inhalt unerlässlich. Ausserdem kann davon ausgegangen werden, dass das Datenvolumen der Metadatenbank beschränkt ist. Aus diesen Gründen wird beim Arbeiten mit einer Datenbank ihre Beschreibung redundant im Hauptspeicher gehalten. Die Komponente Description Manager übernimmt diese Verwaltung.

3.7 Prozedurale Schnittstelle mit erweitertem Cursorkonzept

Die prozedurale Schnittstelle erlaubt das Verarbeiten von K-, H- oder M-Tupeln, die versionenbehaftet sein können. Im Gegensatz zur klassischen Eintupelschnittstelle relationaler Datenbanksysteme beziehen sich Abfrage- und Manipulationsoperationen der Schnittstelle XDS auf Mengen von Tupeln (Löschen oder Kopieren von Objekten und Teilobjekten). Natürlich beschränkt sich ein prozeduraler Aufruf für das Einfügen oder Verändern z.B. eines abhängigen Tupels auf einen bestimmten Datensatz, doch erlaubt ein *erweitertes Cursorkonzept* solche Operationen an derselben Stelle oder innerhalb desselben Objekts bzw. derselben Objektmenge zu wiederholen. Dies führt zu ***molekularen Operationen*** auf K-, H- und M-Tupeln sowie beliebigen Versionenmengen.

Bei *Datenbankabfragen* erlaubt die Angabe einer Zugriffsebene und eines Bewegungsmodus, eine beliebige Relation im klassischen Sinne von Datensatz zu Datensatz zu durchlaufen, eine H-Relation in der hierarchischen Reihenfolge abzufragen, Punkt- und Bereichfragen zu beantworten sowie einzelne Versionen oder ganze Versionenmengen zu lesen.

Das Navigieren ist auf den folgenden *Zugriffsebenen* möglich, die sich auf die Feinheit der Cursorbewegung beziehen:

- Schlüssel: Das Navigieren erfolgt innerhalb einer Relation mit oder ohne Angabe eines eventuell nur zum Teil spezifizierten Schlüsselwerts. Wird kein Schlüsselwert angegeben, entspricht die Bewegung dem klassisch-relationalen Zugriff.
- Version: Innerhalb einer Versionenmenge können sämtliche Versionen aufgrund der Versionennummer V# des Surrogats durchsucht werden.
- Versionenmenge: Hier ist das Navigieren mit einem Wechsel der Versionenmenge verbunden, wobei der Cursor nur auf Versionenmengen innerhalb einer bestimmten Relation positioniert werden kann (Versionenmengennummer S#).
- H-Tupel: Diese Zugriffsebene erlaubt es, alle Datensätze eines H-Tupels in der Reihenfolge der physischen Abspeicherung zu durchlaufen, basierend auf den Surrogatteilen Relationennummer R# und Laufnummer N#.
- H-Relation: Das Navigieren zwischen Relationen ist bei H-Relationen wichtig. Damit werden Bewegungen zwischen Tupeln ermöglicht, die nicht derselben abhängigen Relation angehören (Relationennummer R#).

Bei M-Relationen ist es zusätzlich möglich, die zu selektierenden Tupel einer Relation in einer Prozedur zu beschreiben. Durch solche *benutzerdefinierten Prozeduren* können allgemeine Bereichfragen (z.B. Punkte innerhalb eines Polygons) oder Anfragen an zwei Relationen (z.B. Schnittpunkte von Kreisen und Rechtecken) formuliert werden.

Neben der Zugriffsebene ist der *Bewegungsmodus* wichtig, welcher je nach Feinheit der Cursorbewegung eine eigene Bedeutung aufweist:

- Vorhergehendes: Der Cursor wird auf dem vorhergehenden Tupel positioniert, wobei die Feinheit der Bewegung von der gewählten Zugriffsebene abhängt.
- Nächstes: Der Cursor wird auf dem nächsten Tupel positioniert, abhängig von der gewählten Zugriffsebene.
- Erstes: Der Cursor wird auf das erste Tupel innerhalb der gewählten Zugriffsebene positioniert. Auf der Zugriffsebene H-Relation erfolgt die Bewegung auf folgende, vom Wortlaut abweichende Art: Wechsel auf die erste abhängige Relation.
- Letztes: Der Cursor wird gemäss der Zugriffsebene auf das letzte Tupel positioniert. Auf der Zugriffsebene H-Relation erfolgt die Bewegung auf folgende, vom Wortlaut abweichende Art: Wechsel auf die übergeordnete Relation.

Bei *Datenbankmanipulationen* erlaubt die Angabe einer Mutationsebene die Menge der betroffenen Daten zu bestimmen. Es ist möglich, ganze Versionenmengen zu löschen, eine Kopie einer Version anzulegen bevor z.B. Änderungen an ihr vorgenommen werden, aber auch vollständige K-, H- oder M-Relationen (inkl. ihrer Beschreibung in der Metadatenbank) zu eliminieren oder zu duplizieren.

Die *Mutationsebene* kennt folgende Granulate:

- Tupel: Bei der Angabe der Mutationsebene Tupel wird ein K-, H- oder M-Tupel selektiert.

Es ist auch möglich, ein abhängiges Tupel anzugeben.

- Version: Aufgrund des Granulats wird eine Version innerhalb einer Versionenmenge ausgewählt.
- Versionenmenge: Eine Versionenmenge mit all ihren Versionen wird für die auszuführende Operation vorgemerkt.
- Metadatenbank: Eine Operation mit dem Granulat Metadatenbank selektiert eine K-, H- oder M-Relation mit ihrer Beschreibung aus der Metadatenbank und all ihre Ausprägungen. Manipulationen auf dieser Ebene sind mit Vorsicht auszuführen, können sie doch das Löschen einer ganzen Relation zur Folge haben.

Aufgrund des Cursorkonzeptes mit Zugriffsebene, Bewegungsmodus und Mutationsebene lassen sich die Datenbankoperationen wie folgt zusammenfassen (siehe auch Anhang):

Retrieve liefert einen Datensatz, wobei sämtliche Zugriffsebenen und Bewegungsmodi zulässig sind. Ergibt die Anfrage an die Datenbank aufgrund der gewählten Zugriffsebene als Resultat eine Menge von Tupeln (z.B. Zugriffsebene 'Schlüssel' mit unvollständig spezifiziertem Schlüsselwert), so wird je nach gewähltem Bewegungsmodus das erste oder letzte Tupel dieser Menge ausgegeben; die weiteren Tupel sind über die beiden anderen Bewegungsmodi 'Nächstes' und 'Vorhergehendes' anzusprechen.

GeometricSearch erlaubt allgemeine Bereichfragen und Anfragen über geometrische Beziehungen zwischen M-Relationen. Voraussetzung für diese Art von Datenbankabfragen ist, dass die entprechenden Daten in einer Gitterdatei abgelegt sind. Die Bewegungsmodi ermöglichen das Suchen weiterer Datensätze in unterschiedlichen Raumdimensionen oder in benutzerspezifizierten Regionen.

Insert fügt einen Datensatz in die Datenbank ein, der zur Bildung einer neuen Version oder Versionenmenge benützt werden kann. Für das Einfügen eines abhängigen Tupels in ein H-Tupel muss die H-Referenz explizit durch einen Surrogatwert oder implizit durch die Cursorposition gegeben sein, beim Einfügen einer Version in eine Versionenmenge muss die Menge bekannt sein.

Replace ersetzt in einem Datensatz beliebige Merkmalswerte, ausser dem Surrogat und der H-Referenz.

Delete löscht ein Tupel, eine Version, eine Versionenmenge oder eine K-, H- oder M-Relation in der Datenbank. Das Löschen einer ganzen Relation bewirkt eine Modifikation des Datenbankschemas.

Copy dupliziert eine Version innerhalb einer Versionenmenge, eine ganze Versionenmenge innerhalb einer Relation oder eine K-, H- oder M-Relation inklusive ihrer Beschreibung. Dabei werden die betroffenen Surrogatkomponenten neu vergeben, externe vom Benutzer definierte Referenzen werden nicht nachgeführt. Das Kopieren einer ganzen Relation bewirkt eine Erweiterung des Datenbankschemas. Die Operation Copy dient vor allem der Versionenkontrolle, indem sie erlaubt, neue Versionen und Versionenmengen aus alten herzuleiten.

Dieselben Operationen werden auch zur Definition des Datenbankschemas in der Metadatenbank verwendet. Zudem existieren Operationen für das Generieren, Eröffnen, Löschen und Schliessen einer Datenbank.

4. Projektstand und Ausblick

Die Implementierung des Datenbankkerns von XRS basierend auf den Konzepten aus den Abschnitten 2 und 3 ist für verschiedene Rechner abgeschlossen und erlaubt, künftig auch Aspekte des Mehrbenutzerbetriebs, Verfahren zur Konsistenzsicherung oder Konstrukte zur Datenbankabfrage und -manipulation auf einer höheren Sprachebene zu studieren. Wir planen dazu eine Erweiterung von Modula/R [Koch et al. 1983], einer in Modula-2 eingebetteten, deskriptiven Datenbankprogrammiersprache, welche ein Prädikatenkalkül erster Ordnung zur Auswertung von logischen Ausdrücken über Relationen umfasst.

Zwei wichtige Teilprojekte sind zurzeit in Bearbeitung. Das erste dient der Kopplung unseres *geometrischen Modellierers POLY* [Meier/Loacker 1987] mit dem Datenbankkern von XRS, um einerseits über eine anspruchsvolle Testumgebung zu verfügen, andererseits aber auch wichtige Erfahrungen mit H- oder M-Relationen sowie mit Versionen im Bereich rechnergestützter Konstruktion zu sammeln. Die Versionenverwaltung eignet sich z.B. bei der Variantenkonstruktion oder für die Maschengenerierung bei der Finiten Elementberechnung. Neben dem Kopierverfahren zur Versionenbildung wollen wir künftig auch *Differenzverfahren* anbieten [Meier/Petry 1986]; diese sind besonders bei H-Relationen interessant und dank dem gewählten Surrogatkonzept im bereits implementierten Datenbankkern integrierbar. Eine Graphverwaltung soll die Modellierung von Beziehungen zwischen Versionen ermöglichen.

Das zweite Teilprojekt untersucht die Datentypen SURROGATE, TENSOR, SET und SEQUENCE als mögliche Kandidaten zur geplanten Spracherweiterung von Modula/R:

Neben der Einbeziehung des Datentyps SURROGATE zur Definition molekularer Operationen soll ein Datentyp TENSOR angeboten werden [Meier 1986a]. Unter Tensoren versteht man Grössen, die bei bestimmten Transformationen des zugrundeliegenden Koordinatenraumes invariant bleiben. Tensoren nullter Stufe sind Skalare, solche erster Stufe sind Vektoren und als Tensoren zweiter Stufe können z.B. Matrizen dargestellt werden. Die wichtigsten algebraischen Tensoroperationen sind Multiplikation mit einem Skalar, Addition zweier Tensoren derselben Stufe, tensorielles Produkt und Verjüngung. Obwohl solche Tensoren aus dem geometrischen Anwendungsbereich nicht wegzudenken sind, können sie nicht direkt in einer klassischen relationalen Datenbank abgespeichert werden. Der Grund liegt in der Forderung der Ersten Normalform. Zur Speicherung eines dreidimensionalen Vektors sind z.B. drei Attribute für die x-, y- und z-Koordinaten notwendig. Bei der Speicherung von Matrizen oder Tensoren höherer Stufen wird diese Problematik noch offensichtlicher.

Ähnliche Schwierigkeiten ergeben sich auch, wenn man Mengen oder Folgen (z.B. Messreihen) mit Hilfe von Relationen darstellen möchte. Dazu dienen in unserer Erweiterung die beiden Datentypen SET und SEQUENCE, welche das Einfügen und Entfernen von Elementen fest vordefinierter Grundtypen in Mengen und Folgen erlauben oder Nachfolge- und Vorgängerelemente eines gewählten Elementes innerhalb einer Folge bestimmen.

Dank: Viele Ideen und Konzepte von XRS sind mit verschiedenen Fachkollegen diskutiert und verifiziert worden. Für ihre Hilfe danken wir T. Härder, R. A. Lorie, J. Nievergelt, P. Pistor und H.-J. Schek. Zur Verbesserung des vorliegenden Berichtes haben K. Dittrich, R. Laue und P. Pistor durch konstruktive Kritik beigetragen.

LITERATUR

[Biagioni et al. 1986]

Biagioni E., Heiser G., Hinrichs K., Müller C.: OSSI - A Protable Operating System Interface and Utility Library for Modula-2. ETH Zürich, Institut für Informatik, Bericht Nr. 67, Juli 1986.

[Blaser/Pistor 1985]

Blaser A., Pistor P. (Ed.): Datenbank-Systeme für Büro, Technik und Wissenschaft. Informatik-Fachberichte Nr. 94, Springer-Verlag, Berlin 1985.

[Deppisch et al. 1985]

Deppisch U., Obermeit V., Paul H.-B., Schek H.-J., Scholl M., Weikum G.: Ein Subsystem zur stabilen Speicherung versionenbehafteter, hierarchisch strukturierter Tupel. In: [Blaser/Pistor 1985], S. 421-440.

[Dittrich et al. 1985]

Dittrich K. R., Kotz A. M., Mülle J. A., Lockemann P. C.: Datenbankunterstützung für den ingenieurwissenschaftlichen Entwurf. Informatik-Spektrum 8, 1985, S. 113-125.

[Hall et al. 1976]

Hall P., Owlett J., Todd S.: Relations and Entities. In: Nijssen G. M. (Ed.): Modelling in Data Base Management Systems. North-Holland, Amsterdam 1976, pp. 201-220.

[Härder 1978]

Härder T.: Implementierung von Datenbanksystemen. Carl Hanser Verlag, München 1978.

[Härder/Reuter 1985]

Härder T., Reuter A.: Architektur von Datenbanksystemen für Non-Standard-Anwendungen. In: [Blaser/Pistor 1985], S. 253-286.

[Hinrichs 1985]

Hinrichs K.: Implementation of the GRID File: Design Concepts and Experience. BIT 25, 1985, pp. 569-592.

[Koch et al. 1983]

Koch J., Mall M., Putfarken P., Reimer M., Schmidt J. W., Zehnder C. A.: Modula/R Report. Lilith Version. ETH Zürich, Institut für Informatik, Februar 1983.

[Lamersdorf/Schmidt 1983]

Lamersdorf W., Schmidt J. W.: Rekursive Datenmodelle. Informatik-Fachberichte Nr. 83, Springer-Verlag, Berlin 1983, S. 148-168.

[Lorie et al. 1985]

Lorie R.A., Kim W., McNabb D., Plouffe W., Meier A.: Supporting Complex Objects in a Relational System for Engineering Databases. In: Kim W., Reiner D. S., Batory D. S. (Eds.): Query Processing in Database Systems. Springer-Verlag, Berlin 1985, pp. 145-155.

[Lum et al. 1985]

Lum V., Dadam P., Erbe R., Günauer J., Pistor P., Walch G., Werner H., Woodfill J.: Design of an Integrated DBMS to Support Advanced Applications. In: [Blaser/Pistor 1985], pp. 362-381.

[Meier 1986a]

Meier A.: Applying Relational Database Techniques to Solid Modelling. Computer-Aided Design, Vol. 18, No. 6, July/August 1986, pp. 319-326.

[Meier 1986b]
Meier A.: Methoden der grafischen und geometrischen Dateverarbeitung. Teubner Verlag, Stuttgart 1986.

[Meier/Lorie 1983a]
Meier A., Lorie R. A.: A Surrogate Concept for Engineering Databases. Proc. Int. Conf. on VLDB, Florence 1983, pp. 30-32.

[Meier/Lorie 1983b]
Meier A., Lorie R. A.: Implicit Hierarchical Joins for Complex Objects. IBM Research Report RJ3775, San Jose 1983, pp. 1-13.

[Meier/Petry 1986]
Meier A., Petry E.: Versionenkontrolle geometrischer Daten. Eingeladener Vortrag GI-Fachgespräch "Verarbeitung und Verwaltung geometrischer Daten", 16. GI-Jahrestagung Berlin, Informatik-Fachberichte Nr. 126, Springer-Verlag, Berlin 1986, S. 498-512.

[Meier/Loacker 1987]
Meier A., Loacker H.-B.: POLY - Computergeometrie für Informatiker und Ingenieure. McGraw-Hill Verlag, Hamburg, 1987.

[Nievergelt et al. 1984]
Nievergelt J., Hinterberger H., Sevcik K.: The Grid File: An Adaptable, Symmetric Multikey File Structure. ACM Transactions on Database Systems, Vol. 9, No. 1, March 1984, pp. 38-71.

[Schek/Scholl 1986]
Schek H.-J., Scholl M. H.: The Relational Model with Relation-valued Attributes. Information Systems, Vol. 11, No. 2, pp. 137-147, 1986.

[Wirth 1985]
Wirth N.: Programming in Modula-2. Springer-Verlag, Berlin 1985.

[Zehnder 1987]
Zehnder C. A.: Informationssysteme und Datenbanken. Teubner Verlag, Stuttgart 1987.

Anhang: Syntax und Semantik der XDS-Schnittstelle

```
PROCEDURE Retrieve (VAR relName: NameType;
                    key        : RKey;
                    level      : AccessLevel;
                    mode       : NavigationMode;
                    VAR tup    : Tuple;
                    VAR sur    : Surrogate;
                    VAR inf    : ReturnInfo);
```

Die Prozedur `Retrieve` liest Datensätze einer bestimmten Relation aus der Datenbank, aufgrund eines eventuell nur zum Teil spezifizierten Schlüssels und einer zu wählenden Zugriffsebene.

Der Zugriff auf Datensätze auf den Zugriffsebenen (**level**) Versionenmenge, Version, H-Tupel und H-Relation erfolgt stets über das Surrogat; die Zugriffsebene Schlüssel bietet die Selektion von Datensätzen über verschiedene Schlüsseltypen an, deren Werte nur zum Teil spezifiziert zu sein müssen.

Neben der Zugriffsebene kann die Art des Navigierens auf den Datensätzen gewählt werden (**mode**). Das Navigieren erfolgt stets bezüglich eines Referenzpunktes; als Referenzpunkt wird ein Surrogatwert akzeptiert.

Der Referenzpunkt und der mögliche Schlüsseltyp werden im Parameter **key** spezifiziert. Zur Verfügung stehen drei Schlüsseltypen: Surrogat, mehrdimensionaler Schlüssel und Benutzerschlüssel. Das Surrogat verkörpert den Identifikationsschlüssel, ist also eindeutig und zu jedem Datensatz vorhanden. Ein mehrdimensionaler Schlüssel kann nur zur Selektion von Datensätzen einer M-Relation benutzt werden. Im Gegensatz zum Surrogat und mehrdimensionalen Schlüssel sind mehrere Benutzerschlüssel zu einer Relation möglich. Der Zugriff über Attribute oder Attributkombinationen, für die keine Zugriffspfade existieren, ist ebenfalls möglich.

Für den klassisch-relationalen Zugriff wählt man die Zugriffsebene Schlüssel mit Schlüsseltyp Surrogat und spezifiziert beim Schlüsselwert nur den Surrogatteil Relationennummer. Das Resultat dieser Anfrage ist eine Menge von Datensätzen einer Relation, auf der navigiert werden kann. Insbesondere kann sie sequentiell nach einer systeminternen Reihenfolge durchlaufen werden.

Die Prozedur `Retrieve` liefert den Datensatz (**tup**) und den Surrogatwert **sur** des Tupels (als Kopie des entsprechenden Attributwerts), der als Referenzpunkt für eventuell folgendes Navigieren dienen kann, sowie eine Rückmeldung (**inf**). Letztere teilt dem Anwender resp. aufrufenden Programm nicht nur mit, ob die Operation erfolgreich durchgeführt werden konnte, sondern bietet auch eine Information für das Navigieren an. Da auf den Zugriffsebenen H-Tupel und H-Relationen ein Wechsel der Relation möglich ist, wird in **relName** der Name der Relation ausgegeben zu der das Tupel gehört.

```
PROCEDURE GeometricSearch (key     : GKey;
                           mode    : NavigationMode;
                           VAR tup : TupPair;
                           VAR sur : SurPair;
                           VAR inf : ReturnInfo);
```

Die Prozedur `GeometricSearch` dient dem Lesen von Datensätzen aus M-Relationen. Im Gegensatz zu `Retrieve` kann aber die Anfrage eine Resultatsmenge von Datensatz*paaren* mit je einem Datensatz aus zwei verschiedenen Relationen ergeben. Zudem bezieht sich `GeometricSearch` stets auf den Zugriff über einen mehrdimensionalen Schlüssel. `GeometricSearch` dient hauptsächlich der Beantwortung von Punkt-, Teilpunkt-, Bereich- und Teilbereichfragen, die sich auch auf zwei Relationen ausdehnen lassen.

In den meisten Fällen wird eine Anfrage mit `GeometricSearch` an die Datenbank eine Resultatsmenge ergeben. Auf welches Element der Resultatsmenge als erstes resp. nächstes zugegriffen werden soll, kann durch Angabe des Bewegungsmodus (**mode**) festgelegt werden.

Anfragen an die Datenbank mit GeometricSearch lassen sich in zwei Gruppen einteilen: das Lesen von Datensätzen aus einer Relation oder das Extrahieren von Datensatzpaaren aus zwei Relationen, die zueinander in einer bestimmten Beziehung (wie Inklusion oder Schnitt) stehen. Die Art der Anfrage, die Namen der betroffenen Relationen, der Referenzpunkt für das Navigieren innerhalb der Resultatsmenge sowie die Spezifikation der Schlüsselwerte geschehen im Parameter **key**. Bei der Bereichfrage können die zu selektierenden Datensätze in einer Prozedur beschrieben werden, was diesem Anfragetyp wesentlich mehr Flexibilität verleiht als bei einer Anfrage mit Hilfe der Operation Retrieve. Beim Suchen von Datensatzpaaren aus zwei Relationen muss neben den beiden Suchregionen die geometrische Beziehung in einer Prozedur formuliert werden.

Die Prozedur GeometricSearch liefert je nach Anfragetyp einen Datensatz mit zugehörigem Surrogat bzw. Paare von Datensätzen (**tup**) mit zugehörigen Surrogaten (**sur**). Ebenfalls zurückgeliefert wird eine Rückmeldung (**inf**).

```
PROCEDURE Insert (relName : NameType;
                  hRef    : HReference;
                  tup     : Tuple;
                  level   : MutationLevel;
                  VAR sur : Surrogate;
                  VAR inf : ReturnInfo);
```

Insert fügt einen Datensatz in eine Relation der Datenbank ein. Damit können neue Versionen oder Versionenmengen gebildet und abhängige Tupel in H-Tupel eingefügt werden. Über Einfügungen in Relationen des Metadatenbankschemas wird das Datenbankschema definiert.

Die bezeichnete Relation (**relName**) und die gewählte Mutationsebene (**level**) stehen in enger Beziehung zueinander: Bei K-, H- oder M-Relationen dienen die beiden Mutationsebenen Versionenmenge bzw. Version der Bildung einer neuen Versionenmenge *und* ersten Version mit dem angegebenen Tupel bzw. einer neuen Version innerhalb einer bereits bestehenden Versionenmenge. Abhängige Tupel können durch die entsprechende Mutationsebene in abhängige Relationen eingefügt werden. Soll das Datenbankschema erweitert werden, wird eine Relation des Metadatenbankschemas spezifiziert. Die semantische Korrektheit der Schemaerweiterung wird dabei gewährleistet.

Die Referenz zum übergeordneten Tupel (**hRef**) kommt nur bei abhängigen Relationen zum Tragen, nämlich beim Einfügen eines Datensatzes in eine abhängige Relation. Die hierarchische Referenz ist der vollständig spezifizierte Surrogatwert des übergeordneten Tupels.

Zum Zeitpunkt des Einfügens ist der Surrogatwert des Datensatzes (**tup**) dem Benutzer noch nicht bekannt, da der Identifikationsschlüssel vom System vergeben wird. Das entprechende Feld im Tupel kann deshalb unspezifiziert übergeben werden. Insert wird nach erfolgreich abgeschlossener Operation den zugeteilten Surrogatwert nicht nur im Parameter **sur** zurückgeben, sondern ihn auch an der entsprechenden Stelle im Datensatz einsetzen.

Die Prozedur Insert informiert den Benutzer über die durchgeführte Manipulation durch Rückgabe eines Parameters **inf**. Falls die Manipulation nicht ausgeführt werden konnte, kann die genaue Fehlermeldung mittels der speziellen Operation CheckDB erfahren werden.

```
PROCEDURE Copy (relName    : NameType;
                sur        : Surrogate;
                level      : MutationLevel;
                VAR newSur : Surrogate;
                VAR inf    : ReturnInfo);
```

Die Prozedur Copy dient dem Duplizieren einzelner Datensätze oder von Mengen von Datensätzen, wobei bei H-Relationen alle hierarchisch abhängigen Datensätze dupliziert werden. Die Kopieroperation ist auf

jeder Mutationsebene anwendbar, insbesondere auf abhängigen Tupeln und der Metadatenbank. Copy dient hauptsächlich der Versionenverwaltung, d.h. dem Anlegen von Versionen eines Objektes (Duplizieren des Objektes), z.B. um den Stand einer Arbeit festzuhalten, bevor daran weitergearbeitet wird.

Mit der Mutationsebene (**level**) wird spezifiziert, ob eine ganze Versionenmenge, eine Version und abhängige Tupel oder einzelne Tupel dupliziert werden sollen. Duplizieren auf der Metadatenbank ist ebenfalls erlaubt, sofern die semantische Korrektheit damit nicht verletzt wird. So lassen sich insbesondere ganze Relationen kopieren.

Das zu duplizierende und in **sur** bezeichnete Tupel muss der Relation mit dem Namen **relName** angehören. Die mit Copy duplizierten Datensätze erhalten automatisch neue Surrogatwerte zugeteilt. H-Referenzen werden konsistent gehalten. Der Surrogatwert des mit sur bezeichneten und nun duplizierten Datensatzes wird in **newSur** zurückgegeben.

Über die korrekte Ausführung der Operation informiert wieder der Parameter **inf**.

```
PROCEDURE Delete (relName  : NameType;
                  sur      : Surrogate;
                  level    : MutationLevel;
                  VAR inf  : ReturnInfo);
```

Die Prozedur Delete ist in gewisser Weise die inverse Operation zu Copy: Werden bei Copy alle abhängigen Tupel mitdupliziert, so werden sie bei Delete alle gelöscht (cascaded deletion). Es stehen wieder die verschiedenen Mutationsebenen zur Verfügung, was auch das Löschen ganzer Relationen beinhaltet.

Ein Aufruf der Prozedur Delete muss in **relName** den Namen der Relation enthalten, in der das Tupel mit dem Surrogat **sur** gelöscht werden soll. Die Mutationsebene **level** spezifiziert wieder die Granularität der Operation: So lässt sich ein abhängiges Tupel (mit evtl. von diesem abhängigen Tupeln), mit allen enthaltenen Versionen löschen. Löschen ganzer Relationen (K, M oder H) erfolgt über das Löschen des entsprechenden Tupels in der Relation 'Relation'. Daten, Zugriffshilfen und Beschreibung der Relation werden gelöscht. Löschen in Beschreibungen von Relationen oder von abhängigen Relationen ist nur dann erlaubt, wenn die Relation noch keine Daten enthält, da sonst umfangreiche Reorganisationen ausgeführt werden müssten.

Der Rückgabeparameter **inf** enthält wieder Information über die Ausführung der Operation.

```
PROCEDURE Replace (relName  : NameType;
                   sur      : Surrogate;
                   newTup   : Tuple;
                   VAR inf  : ReturnInfo);
```

Replace dient dem Ändern beliebiger Datenwerte, ausgenommen das Surrogat und die eventuell vorhandene H-Referenz.

Die Struktur und die Länge der neuen Daten, deren Referenz in **newTup** übergeben werden, muss identisch sein mit derjenigen des in **sur** angesprochenen Datensatzes der Relation mit Namen **relName**. Die Operation ist eine reine Ein-Tupel-Operation und wirkt wieder gleichermassen auf den Benutzerdaten oder der Metadatenbank.

Zurückgegeben wird im Parameter **inf** wieder der Status der Ausführung.

Konzept und Realisierung einer Versionenverwaltung in einem relationalen Datenbankkern

Erwin Petry

Institut für Informatik
ETH Zürich

Zusammenfassung:
Wir stellen ein erweitertes Relationenmodell zur Verbesserung der Modellierungsmöglichkeiten in neueren Anwendungsgebieten von Datenbanksystemen vor. Das Modell beinhaltet einen Objekt- und einen Versionenbegriff. Anschliessend gehen wir auf die Realisierung der Konzepte innerhalb des Datenbankkerns von XRS ein. XRS ist ein an der ETH Zürich entwickeltes Pilot-Datenbanksystem. Der Datenbankkern ist anwendungsneutral und umfasst alle Basisfunktionen eines Datenbanksystems. Hinzu kommen eine objekt- und versionenbehaftete Sicht der Daten und Möglichkeiten zur Effizienzsteigerung. Über ein Surrogatkonzept und Referenz-Attributtypen wird der Objektzusammenhang dargestellt. Verschiedene Relationentypen ermöglichen es dem Benutzer des Datenbankkerns, die Speicherung seiner Daten zu beeinflussen und so die Effizienz zu verbessern. Der anwendungsneutralen Realisierung einer Versionenverwaltung dienen Graphen. Geringerer Speicherplatzbedarf für Versionen kann durch Differenzbildung erreicht werden. Wir stellen diese Konzepte und deren Realisierung innerhalb von XRS vor. XRS integriert damit Konzepte zur Objektmodellierung, zur Versionenverwaltung und zur Effizienzsteigerung.

Schlüsselwörter:
objektorientiertes Datenmodell, Datenbankkern, Surrogatkonzept, Effizienzsteigerung, Version, Versionengraph, Differenzverfahren.

Summary:
We present an extended relational model to improve modelling capabilities in new application fields of database systems. The model comprises an object and a version term. Afterwards we go into the realization of the ideas within the database kernel of XRS. XRS is a pilot database system developed at ETH Zürich. The kernel is independent of application and includes all basic functions of a database system. An object- and version-oriented view of data and capabilities to improve efficiency are added. A surrogate concept and reference attribute types are used to express the connection inside of objects. With various relation types the user of the database kernel has influence on the storage structures and like this he can improve efficiency. To realize an application independent version management we use graphs. A reduction of used storage is achieved by differences of versions. We present these concepts and their realization within XRS. Consequently XRS integrates concepts for modelling objects, for version management and improvement of efficiency.

Keywords:
objectoriented Datamodel, Databasekernel, Surrogate, Efficiency, Version, Versiongraph, Differenceversion.

Computing Reviews Classification: E.2, H.2, J.6

1. Einleitung

Aus vielen alten und neuen Anwendungsgebieten von Datenbanksystemen [BP 85] kommt der Wunsch, mehrere Versionen von Benutzerdaten vom Datenbanksystem verwalten zu lassen. Bei temporalen Datenbanken geht es primär darum, dass mit den Daten Zeiten assoziiert werden und der Benutzer nach dem Inhalt der Datenbank zu einem bestimmten Zeitpunkt fragen kann [vgl. Hä 84]. Bei technischen Datenbanken gehen die Anforderungen eher dahin, dass grosse, kompliziert strukturierte Objekte mehrfach gespeichert sein sollen, unter Umständen in nur sehr leicht veränderter Form. Für den Benutzer bestehen hier nicht nur zeitliche, sondern auch andersartige Abhängigkeiten zwischen den Versionen seines Objektes und einzelne Versionen haben für ihn bestimmte Eigenschaften. Die Beziehungen zwischen den Versionen können in den einzelnen Anwendungsgebieten von Datenbanksystemen sehr unterschiedlicher Art und Semantik sein [vgl. DL 85].

Neben der Versionenproblematik ist vor allem auch die angemessene Modellierung stark strukturierter Benutzerdaten nicht hinreichend gelöst. Existierende Modelle sind zwar mit Einschränkungen einsetzbar, jedoch geht dabei für effizienzsteigernde Massnahmen benötigtes Kontextwissen verloren. Heutige Datenbanksysteme haben daher in neuen Anwendungsgebieten ein prohibitiv schlechtes Leistungsverhalten.

Zur Lösung dieser und der anderen Probleme, die die neueren Anwendungen von Datenbanksystemen mit sich bringen, geht man vorzugsweise den Weg, dass man versucht, einen Datenbankkern zu implementieren, der anwendungsneutral und möglichst vielseitig ist, um darauf anwendungsspezifische Teile einfach und effizient zu realisieren [Mi 84]. Als ein sehr geeignetes Datenmodell hat sich das Relationenmodell erwiesen, weshalb wir es in erweiterter Form für den hier vorzustellenden Datenbankkern übernehmen wollen. Hinzu kommen Möglichkeiten zur allgemeinen Versionenverwaltung und zur Effizienzsteigerung der Datenhaltung.

Im nächsten Kapitel wird ein Modell zur Objektmodellierung und zur Versionenverwaltung vorgestellt. Im dritten Kapitel wird dann auf die Einbettung des Modells in den Datenbankkern von XRS eingegangen und im vierten Kapitel die Implementierung des Datenbankkerns besprochen. Speziell wird dabei die Schnittstelle und die Implementierung von Graphen und Differenzversionen erklärt.

2. Ein Modell zur Objektmodellierung und Versionenverwaltung

Beim Schemaentwurf für eine relationale Datenbank sucht man nach Objekten, die durch einige Attribute beschreibbar sind. Ein Tupel einer Relation repräsentiert dann das Objekt in der Datenbank. Alle Objekte, die durch die gleichen Attribute in der Datenbank repräsentiert werden, werden in die gleiche Relation aufgenommen und sind dort die einzelnen Tupel. Für verschiedenartige Objekte gibt es dann mehrere Relationen in der Datenbank. Wir wollen die Objekte, die für einen Benutzer einen natürlichen Zusammenhang haben, die Einheiten in denen der Benutzer denkt, als Benutzerobjekte bezeichnen. Die Modellierung im Relationenmodell geht solange in sehr natürlicher Weise, wie die Benutzerobjekte sich durch eine feste, im voraus bekannte Anzahl atomarer Attribute darstellen lassen und keine komplizierten Beziehungen zwischen den Objekten bestehen. Um eine grössere Klasse von Benutzerobjekten, die diese einschränkenden Eigenschaften nicht unbedingt besitzen, dennoch im Relationenmodell zu modellieren, ist es

notwendig, mehrere Relationen zur Beschreibung einzelner Benutzerobjekte zu verwenden. Beispiele für solche Objekte lassen sich in allen neuen Anwendungsbereichen für Datenbanken finden. Zu erwähnen sind beispielsweise der VLSI-Chip und das Maschinenteil. Es sind dies die viel diskutierten "komplexen Objekte" [Mi 85].

Wir führen darum eine weitere Strukturierungsmöglichkeit für die zu speichernden Daten ein, nämlich die Objektklasse. Eine Datenbank besteht i.a. aus mehreren Objektklassen, eine Objektklasse umfasst mehrere Relationen. Eine Relation hat wie im Relationenmodell mehrere atomare Attribute und einen Identifikationsschlüssel. Die Strukturierung soll bei der Schemadefinition derart vorgenommen werden, dass all diejenigen Relationen zur gleichen Objektklasse gehören, die nötig sind, um ein Benutzerobjekt abzulegen.

Ein Beispiel für eine solche Objektklasse sind die fünf Relationen aus Bild 2.1, mit denen sich ebenbegrenzte, dreidimensionale Körper darstellen lassen. Eine Ansammlung von Tupeln aus all diesen Relationen repräsentiert ein Benutzerobjekt, nämlich einen komplizierten Körper, wie z.B. ein bestimmtes Maschinenteil. Eine solche Ausprägung in einer Objektklasse bezeichnen wir als Objekt. Zur Modellierung der inneren Struktur komplizierter Objekte wird hier das Relationenmodell eingesetzt. Im Beispiel werden der Identifikationsschlüssel der Relation `vertex` (`vertex#`) in der Relation `edge` als Fremdschlüssel verwendet (`startvertex#` und `endvertex#`), womit Punkte und Kanten einander zugeordnet werden. Attribute, die diese inneren Beziehungen tragen, müssen von einem speziellen Typ sein, damit das Datenbanksystem die Zusammengehörigkeit eines Objektes kontrollieren kann (in Bild 2.1 noch unberücksichtigt).

```
solid (body   (name: NameType);
       face   (face#: CARDINAL; bodyname: NameType);
       ring   (ring#, face#: CARDINAL);
       edge   (leftring#, rightring#, startvertex#, endvertex#: CARDINAL);
       vertex (vertex#: CARDINAL; body: NameType; x, y, z: REAL)          )
```

Identifikationsschlüssel sind unterstrichen.

Bild 2.1: Die Objektklasse `solid` für ebenbegrenzte, dreidimensionale Körper

Für Objekte mit einer anderen Struktur, wie z.B. VLSI-Chips, muss auch eine andere Objektklasse definiert werden. Sind die Objekte "einfach", genügt vielleicht bereits eine einzige Relation. Das Relationenmodell mit seiner Intention, gleichartige Benutzerobjekte in einer Relation zu speichern, stellt also einen Spezialfall dieses Modells dar. Es ist uns damit möglich, Benutzerobjekte sehr verschiedener Komplexität und unabhängig vom Anwendungsgebiet zu modellieren.

Ein Benutzerobjekt besteht jetzt nicht wie bisher aus einem Tupel einer Relation, sondern aus Tupeln verschiedener Relationen. Aus jeder Relation können mehrere Tupel zum gleichen Objekt gehören. Jedes Tupel in der Datenbank gehört damit sowohl zu einer Relation als auch zu einer Objektklasse, beziehungsweise ist es Teil eines Objektes.

Bei der Versionenverwaltung geht es sowohl bei temporalen wie bei technischen Datenbanken darum, dass jedes Benutzerobjekt mehrfach vorliegt. Dies bedeutet, dass das Datenbanksystem die Daten, die das Objekt repräsentieren, in mehr oder weniger geänderter Form mehrfach verwalten muss. Wir sprechen dann von mehreren Versionen des Objektes. Die Gesamtheit aller Versionen eines Objektes bezeichnen wir als

Versionenmenge.

Mit dieser Erweiterung gehört jedes Tupel in der Datenbank dann zusätzlich noch zu genau einer Version. Wir haben damit zwei Betrachtungsweisen der Daten in einer Datenbank, die im Bild 2.2 verdeutlicht werden. Eine Datenbank besteht aus mehreren Objektklassen (VLSI-Chips, Software-Module, dreidimensionale Körper), die jeweils mehrere Versionenmengen umfassen (Fahrgestell, Karosserie). Eine Versionenmenge enthält mehrere Versionen (Modell A, Modell B der Karosserie) und jede Version wird repräsentiert durch eine Reihe von Tupeln. Daneben besteht aber auch die gleiche Datenbank aus vielen Relationen mit Tupeln. Jede Relation enthält aber nur einen Teil der Daten, die eine Version des Benutzerobjektes repräsentieren.

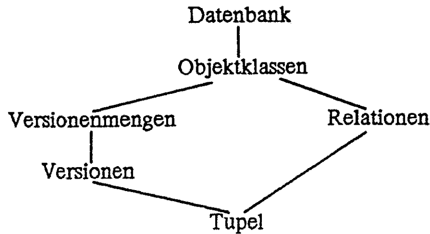


Bild 2.2: Zugehörigkeiten von Datenmengen in einer Datenbank

Bei der Versionenverwaltung im Datenbanksystem soll es nun darum gehen, dem Benutzer einerseits Manipulations- und Zugriffsmöglichkeiten auf Versionen zu bieten, andererseits die Speicherung und den Zugriff auf die Daten effizient zu gestalten.

Wie erwähnt, bestehen zwischen Versionen unterschiedliche Beziehungen, die in einem Datenbankkern aber neutral darzustellen sein sollten, ohne auf einzelne Anwendungsklassen ausgerichtet zu sein. Wir greifen deshalb auf Graphen zurück, die dies genau tun. Bei temporalen Datenbanken liegt ein linearer Graph vor von der ältesten Version bis zur aktuellen oder eventuellen zukünftigen Versionen. Eigenschaften einzelner Versionen sind hier beispielsweise, ob eine Version rechtsgültig ist oder nicht. Bei technischen Datenbanken können die Versionen einer Versionenmenge beispielsweise einen Baum bilden. Die Wurzel repräsentiert dann etwa eine Version, aus der mehrere andere abgeleitet wurden, eventuell über mehrere Zwischenversionen. Eigenschaften können hier beispielsweise sein, ob eine Version, die zur Produktion des entsprechenden Teils gültige Version ist oder ob sie nur einen Versuch für eine Konstruktion darstellt.

Wir führen deshalb in unser Modell der Versionenverwaltung die Möglichkeit ein, einen beliebigen Graphen auf den Versionen einer Versionenmenge zu definieren. Die Knoten des Graphen sind die Versionen; die Kanten sind gerichtet vom Vorgänger zum Nachfolger. Jede Version hat keinen, einen oder mehrere Vorgänger und Nachfolger aus der gleichen Versionenmenge (inkl. sich selbst). Der Graph ist damit nicht notwendigerweise zusammenhängend. Die Bedeutung dieser logischen Vorgänger-/Nachfolgerbeziehung bleibt ausschliesslich Sache des Anwenders; insbesondere interpretiert das Datenbanksystem keine zeitliche Beziehung. Unser angestrebter Datenbankkern muss Operationen zum Verändern und Navigieren auf diesem Graphen anbieten.

3. Unterstützung von Versionenverwaltung im XRS-Modell

Die folgenden Kapitel widmen sich der konkreten Realisierung des eingeführten Modells innerhalb des Datenbankkerns von XRS. Dazu wird zunächst das XRS-Modell an der Kernschnittstelle unter besonderer Berücksichtigung der Versionenverwaltung vorgestellt.

XRS (eXtendend Relational System) ist der Name eines Pilot-Datenbanksystems, das am Institut für Informatik der ETH Zürich entwickelt wird [XRS 87]. Zur Zeit existiert eine lauffähige Version einer prozeduralen Kernschnittstelle, die nun überarbeitet und spezieller auf die Versionenverwaltung ausgerichtet wird. Die Ziele, die dabei im Vordergrund stehen, sind einerseits das Anbieten des im vorigen Kapitel beschriebenen Modells und andererseits die effiziente Verwaltung der Daten.

3.1 Ein Surrogatkonzept

Im klassischen Relationenmodell dürfen zur eindeutigen Identifizierung eines Tupels keine zwei Tupel mit gleichen Werten auf einem(r) bestimmten Attribut(kombination), dem Identifikationsschlüssel, in einer Relation gespeichert werden. Für uns gibt es zur Lösung des Identifikationsproblems eine Reihe von Gründen, einem Ansatz von Hall, Owlett und Todd [HOT 76] zu folgen und jedem Tupel einen systemkontrollierten Bezeichner, ein sogenanntes Surrogat, zuzuordnen. Erstens sind damit auf den Datenwerten eines Tupels Redundanzen möglich, was vielfach erwünscht ist, zweitens wird der Benutzer vom sehr lästigen Vergeben von eindeutigen Nummern befreit und drittens geben wir dem Surrogat eine Struktur, die unsere Versionenverwaltung sehr wirksam unterstützt. Ein Surrogat wird einmal beim Einfügen jedes Tupels vom System vergeben. Sein Wert ist in der ganzer Datenbank eindeutig. Es kann vom Benutzer zum Ausdrücken von Beziehungen verwendet werden, darf aber nicht verändert werden und es wird auch nach Löschen des Tupels nicht wieder vergeben.

Im XRS umfasst jeder Surrogatwert vier Teile:

R#: Eine Nummer der Relation (Relation#), zu der das Tupel gehört. Da jede Relation Teil einer Objektklasse ist, impliziert diese Nummer auch die Objektklasse.

S#: Die Nummer der Versionenmenge (Set#) innerhalb der Objektklasse.

V#: Die Nummer der Version (Version#) innerhalb der Versionenmenge.

N#: Eine Laufnummer (sequence Number) innerhalb der Tupel, die zur gleichen Relation, Versionenmenge und Version gehören, da jede Relation mehrere Tupel einer Version enthalten kann.

Bei der Surrogatvergabe erhalten alle Tupel einer Relation die gleiche R#, alle Tupel aller Versionen einer Versionenmenge die gleiche S# und alle Tupel einer Version die gleiche V#. Relationennummern sind innerhalb der Datenbank eindeutig, Versionenmengennummern innerhalb der Objektklasse und Versionennummern innerhalb der Versionenmenge. Unabhängig vom Surrogat führen wir für jede Objektklasse eine Nummer (O#) ein. Dann wird eine Version eindeutig identifiziert durch die Konkatenation von O#, S# und V#.

3.2 Erweiterte Attributtypen

In einer ersten Näherung definieren wir jetzt analog Kapitel 2, dass eine Datenbank aus Objektklassen besteht und jede Objektklasse eine oder mehrere Relationen umfasst (Bild 3.1; vgl. Beispiel in Bild 2.1). Jede Relation ist eine Teilmenge des kartesischen Produkts aus genau einem Surrogat (= ein Attribut vom Typ SURROGATE) und n (0, 1, 2, ...) weiteren Attributen. Als Attributtypen sind alle bekannten, atomaren Grundtypen zulässig, sowie die Typen REFERENCE und VREFERENCE [vgl. LP 83]. Die Werte dieser beiden Referenztypen sind Surrogate. Attribute dieser Typen dienen der Referenzierung anderer Tupel der Datenbank. Beim Typ REFERENCE sind alle Werte zulässig; das System führt keinerlei Prüfungen auf Existenz des referenzierten Tupels durch. Anders beim Typ VREFERENCE, der der Referenzierung innerhalb einer Version dient. Beim Speichern eines Tupels oder Ändern des Attributwerts einer V-Referenz wird überprüft, ob deren R# zur gleichen Objektklasse gehört und ob S# und V# identisch dem eigenen Surrogatwert sind. Damit ist gewährleistet, dass nur ein Tupel der gleichen Version referenziert werden kann. Die Existenz eines Tupels mit dem Surrogat wird nicht überprüft, da sonst das (sequentielle) Einfügen von Tupeln die sich gegenseitig referenzieren, nicht möglich wäre.

Zusätzlich gibt es einen Attributtyp TIMEPOINT. Der Attributtyp ist ebenfalls strukturiert (siehe Anhang) und kann im Zusammenhang mit der Versionenverwaltung zur Zuordnung von Zeitpunkten zu Versionen sehr vorteilhaft eingesetzt werden. Der Typ erspart die Definition mehrerer Attribute zur Darstellung einer Zeit oder Konvertierungen grosser Zahlen in eine strukturierte Darstellung eines Kalendersystems durch den Anwender. TIMEPOINT-Werte sind wie Skalare geordnet.

Geplant ist die Erweiterung der zulässigen Attributtypen um einen Typ TENSOR. Tensoren spielen in technischen Anwendungen eine grosse Rolle. Vektoren und Matrizen lassen sich beispielsweise dann direkt verwalten, was z.B. die umständliche Definition einer 6 x 3 Matrix als 18 Attribute erspart. Sehr vorteilhaft ist vor allem die Integration der Tensoralgebra in die Relationenalgebra [Po 86].

```
solid (body   (b#: SURROGATE; name: NameType;
               creationTime: TIMEPOINT; designer: REFERENCE);
       face   (f#: SURROGATE; body: VREFERENCE);
       ring   (r#: SURROGATE; face: VREFERENCE);
       edge   (e#: SURROGATE;
               leftring, rightring, startvertex, endvertex: VREFERENCE);
       vertex (v#: SURROGATE; body: VREFERENCE; koord: TENSOR(1;3))     )
```

Bild 3.1: Die Objektklasse `solid` für ebenbegrenzte, dreidimensionale Körper im XRS-Modell (erste Näherung)

3.3 Referenzielle Integrität für hierarchische Beziehungen

Eine besondere Rolle innerhalb der möglichen Klassen von Beziehungen zwischen Relationen spielen hierarchische Beziehungen [LP 83]. Eine hierarchische Beziehung liegt immer dann vor, wenn zu einem Tupel einer übergeordneten Relation kein, ein oder mehrere Tupel einer untergeordneten Relation gehören, ein Tupel der untergeordneten Relation aber nicht ohne ein Tupel der übergeordneten Relation existieren kann. Zu denken

ist hier insbesondere an die Aggregation und Generalisierung in [SS 77] und die Strukturen PART-OF und IS-A in [Mei 86].

Die grosse Bedeutung der hierarchischen Beziehungen rechtfertigt die Integration entsprechender Konstrukte in das XRS-Modell. Wie im folgenden gezeigt wird, erhalten wir dadurch eine stärkere semantische Integrität der gespeicherten Daten und erhöhen wesentlich die Zugriffsgeschwindigkeit.

Neben den oben eingeführten Relationen, die wir im folgenden als klassische Relationen (K-Relationen) bezeichnen, führen wir deshalb als weiteren Relationentyp die hierarchische Relation (H-Relation) ein (siehe Bild 3.2). Eine H-Relation ist ein Baum von Relationen, mit genau einer Wurzelrelation und abhängigen Relationen. Jede Relation dieses Baums kann weitere abhängige Relationen haben. Relationen der nächst höheren Stufe werden übergeordnete Relationen genannt. Analog sprechen wir von einem Wurzeltupel, abhängigen und übergeordneten Tupeln. Für jedes Tupel aus einer Relation innerhalb der H-Relation gilt, dass es kein, ein oder mehrere abhängige Tupel aus jeder der abhängigen Relationen haben kann, dass aber ein abhängiges Tupel nicht ohne sein übergeordnetes Tupel existieren kann. Bei einem Tupel der Wurzelrelation mit all seinen abhängigen Tupeln sprechen wir von einem H-Tupel. Ein abhängiges Tupel mit den von ihm abhängigen Tupeln wird als Teiltupel bezeichnet.

Eine H-Relation muss vollständig zur gleichen Objektklasse gehören, ebenso wie ein H-Tupel vollständig zu genau einer Version gehören muss. Es können mehrere H-Tupel zur gleichen Version gehören.

Für jede abhängige Relation muss genau ein Attribut vom Typ HREFERENCE definiert werden. Der Wert dieses Attributs in einem abhängigen Tupel ist der Surrogatwert des übergeordneten Tupels. Das System ist voll verantwortlich für die Überwachung der referenziellen Integrität. Hier liegt also ein wesentlicher Unterschied zum Definieren mehrerer K-Relationen mit Attributen vom Typ VREFERENCE. Der Benutzer definiert immer genau dann sinnvollerweise eine H-Relation, wenn zwischen mehreren Relationen (u.a.) eine hierarchische Beziehung besteht, die er einer besonderen Systemkontrolle unterziehen möchte.

Im Bild 3.2 ist unser Beispiel dem vollständigen XRS-Modell angepasst und erweitert worden. Umfasste das Beispiel bis jetzt die Darstellung dreidimensionaler Körper in Randdarstellung (boundary representation), ist jetzt die Darstellung der Entstehungsgeschichte des Objektes hinzugekommen. Es handelt sich dabei um das CSG-Konstruktionsprinzip [Re 80] bei dem komplizierte Körper aus Primitivkörpern mittels mengentheoretischer Operatoren (Vereinigung, Durchschnitt, Differenz) konstruiert werden. Zu einer Version eines Körpers gehören dann ein Tupel in der Relation `object`, ein H-Tupel in der Relation `boundaryRepresentation` (die Randdarstellung) und mehrere H-Tupel in der Relation `csgNode`, die jeweils einen Knoten des Konstruktionsbaums beschreiben. Auf die Relation `vertex` wird weiter unten eingegangen. (Das Beispiel ist das leicht veränderte Schema einer XRS-Datenbank [Mel 86], die dem langfristigen Abspeichern der Körper dient, die mit dem dreidimensionalen Modellierer POLY [POLY 86] erzeugt werden.)

Mit der Einführung von Surrogaten, ihrer speziellen Struktur und den Referenz-Attributtypen haben wir die Möglichkeit geschaffen, stark strukturierte Objekte in einer relationalen Datenbank darzustellen. Die Unterscheidung dreier verschiedener Referenztypen ist wesentlich beeinflusst davon, referenzielle Überprüfungen bei Änderungen auf der Datenbank ohne aufwendiges Durchsuchen von Daten durchführen zu können.

```
solid (object: KRELATION
          (obj#: SURROGATE; name: NameType;
           creationTime: TIMEPOINT; designer: REFERENCE;
           br, csg: VREFERENCE);
       boundaryRepresentation: HRELATION
          (br#: SURROGATE; object: VREFERENCE);
       face: HRELATION DEPENDENT OF boundaryRepresentation
          (f#: SURROGATE; br: HREFERENCE);
       ring: HRELATION DEPENDENT OF face
          (r#: SURROGATE; face: HREFERENCE);
       edge: HRELATION DEPENDENT OF boundaryRepresentation
          (e#: SURROGATE; br: HREFERENCE;
           leftring, rightring, startvertex, endvertex: VREFERENCE);
       vertex: MRELATION
          (v#: SURROGATE; body: VREFERENCE; koord: TENSOR (1; 3))
          MKEY koord;
       csgNode: HRELATION
          (csgN#: SURROGATE; object: VREFERENCE; t: TENSOR (2; 6, 3));
       combined: HRELATION DEPENDENT OF csgNode
          (comb#: SURROGATE; node: HREFERENCE;
           operation: SetOperation; leftNode, rightNode: VREFERENCE);
       primitive: HRELATION DEPENDENT OF csgNode
          (prim#: SURROGATE; node: HREFERENCE);
       cube: HRELATION DEPENDENT OF primitive
          (c#: SURROGATE; prim: HREFERENCE; xx, yy, zz: REAL);
       torus: HRELATION DEPENDENT OF primitive
          (t#: SURROGATE; prim: HREFERENCE;
           na, ne: CARDINAL; radius, axle: REAL);
       {other primitives}                                              )
```

zur besseren Visualisierung in einer anderen Darstellung:

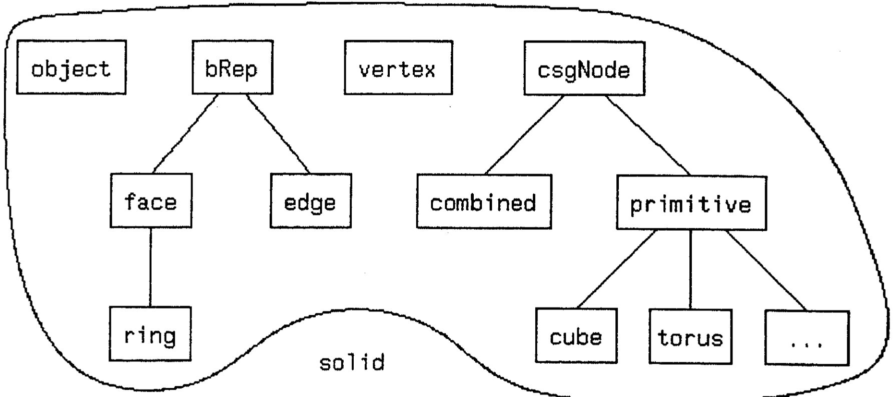


Bild 3.2: Die Objektklasse `solid` für ebenbegrenzte, dreidimensionale Körper in Rand- und CSG-Darstellung im XRS-Modell

3.4 Das XRS-Modell unter Berücksichtigung von Effizienzüberlegungen

Relationale Datenbanksysteme werden heute derart implementiert, dass sie alle Tupel einer Relation physisch clustern, eventuell geordnet nach den Werten bestimmter Attribute. Hinzu kommen Indexstrukturen zum Beschleunigen des Zugriffs nach anderen Attributen. Für die bisherigen Anwendungen relationaler Datenbanken ist diese Vorgehensweise auch sinnvoll, da sie die Anzahl Externspeicherzugriffe für das zu erwartende Zugriffsverhalten minimiert.

Bei technischen Anwendungen mit grossen Benutzerobjekten, die mehrere Relationen einbeziehen, liegt aber ein völlig verändertes Zugriffsverhalten der Anwenderprogramme auf den Daten vor. Es ist zu erwarten, dass sehr häufig alle Tupel einer Version benötigt werden, da beispielsweise ein CAD-System zu einem Zeitpunkt an genau einer Konstruktion, d.h. an einer Version arbeitet. Die Relation tritt zugunsten der Version als Einheit des Zugriffs zurück. Die Relation wird tendentiell nur noch zur Modellbildung herangezogen. Im XRS wird diesen Gegebenheiten durch andere Clusterprinzipien speziell Rechnung getragen.

Als erstes Clusterprinzip werden H-Tupel in preorder-Reihenfolge physisch benachbart abgelegt, was bedeutet, dass ein H-Tupel vollständig auf einer (oder einigen logisch benachbarten) Datenseite(n) gespeichert ist. Die Zugriffszeit auf ein vollständiges H-Tupel wird so minimiert. Es liegen damit auf dieser Datenseite Tupel mehrerer Relationen.

Als zweites Clusterprinzip werden alle Tupel, die zur gleichen Version gehören, physisch benachbart. Dies gilt sowohl in K- als auch in H-Relationen. Tupel aus verschiedenen K- oder H-Relationen werden nicht geclustert. Der Benutzer hat somit durch die Wahl des Relationentyps Einfluss auf die Speicherung. Auf alle Tupel einer Version kann dadurch mi' minimalem Zeitaufwand zugegriffen werden.

Als drittes Prinzip werden alle Versionen einer Versionenmenge geclustert. Dies gil. wiederum nur innerhalb einer K- oder H-Relation.

Als viertes Prinzip lässt sich die Speicherung mehrerer Versionenmengen innerhalb einer Relation auf gleichen Datenseiten betrachten. Dieses Prinzip dürfte in der Praxis allerdings wenig relevant werden, da meist bereits eine Versionenmenge grösser ist als eine Datenseite. Immerhin ermöglicht genau dieses Prinzip die kompakte Speicherung der Daten für den Fall, dass ein Benutzer nicht an einer versionenbehafteten Sicht der Daten interessiert ist und eine Relation lediglich als eine Menge unabhängiger Objekte betrachtet. In einem solchen Fall besteht jede Versionenmenge der Relation aus einem einzigen Tupel. Die Tupel der Relation werden dann auf möglichst wenigen Datenseiten abgelegt.

In CAD-Systemen und geographischen Informationssystemen haben Objekte oder Teile von Objekten oft eine zwei- oder dreidimensionale Position in einem Koordinatensystem. Dadurch ist eine geometrische Nachbarschaft der Objekte definiert. Ist zu erwarten, dass sich der Zugriff auf solche Daten an der geometrischen Nachbarschaft orientiert, ist es sehr wichtig, dass als fünftes Clusterprinzip die Daten gemäss ihrer geometrischen Nachbarschaft auch benachbart im Speicher abgelegt werden.

Das XRS-Modell umfasst deshalb einen dritten Relationentyp, die mehrdimensionale Relation (M-Relation). Sie ist eine spezielle K-Relation, bei der mehrere Attribute als mehrdimensionaler Schlüssel (M-Key) deklariert werden. Die Speicherung der Tupel dieser Relation (M-Tupel) erfolgt physikalisch benachbart nach der Nachbarschaft der Werte des

M-Keys und nicht nach den anderen Clusterprinzipien. Keiner der Schlüsselbereiche wird in irgendeiner Weise einem anderen bevorzugt. Im XRS wird eine solche Relation realisiert über eine Gitterdatei [NHS 84, Hi 85]. In einer solchen M-Relation lassen sich dann Datenbankanfragen mit teilweise spezifiziertem M-Key und Fragen nach allen Tupeln in einem bestimmten Bereich unter Minimierung der Externspeicherzugriffe beantworten.

Im Beispiel ist die Relation `vertex` als M-Relation deklariert mit dem M-Key über dem Koordinatenvektor, der alleine bereits mehrdimensional ist. Die geometrische Nachbarschaft der Punkte bleibt damit im Speicher erhalten. Die Eckpunkte eines Körpers werden so aus der Randdarstellung herausgenommen. Getrennte Manipulation von Topologie und Geometrie eines Körpers ist damit möglich.

Wir haben nun gesehen, dass verschiedene Relationentypen durch verschiedene Speicherprinzipien realisiert werden. Durch die Wahl des Typs kann der Benutzer gezielt Einfluss auf die Speicherung nehmen und so die Ineffizienz relationaler Datenbanksysteme mindern.

3.5 Graphen auf Versionen

Wie im Kapitel 2 eingeführt, beinhaltet das XRS-Modell eine Graphenverwaltung auf den Versionen einer Versionenmenge. Zwischen je zwei Versionen einer jeden Versionenmenge kann eine logische Vorgänger-/Nachfolgerbeziehung (predecessor/successor) bestehen. Den Graphen bezeichnen wir als <u>Präzedenzgraphen</u>. Die Assoziation einer Bedeutung mit den Kanten des Graphen bleibt Sache des Anwenders. Die Graphverwaltung ist also anwendungsneutral.

Wir möchten uns hier darauf beschränken, einen einzigen Graphen auf einer Versionenmenge zuzulassen. Für gewisse Fälle mag es interessant sein, mehrere Graphen definieren zu können [KSW 86]. Ein Graph kann so beispielsweise darstellen, welche Version aus welcher anderen hervorgegangen ist. Ein anderer Graph zeigt die zeitliche Reihenfolge in der die Versionen erstellt wurden. Für die meisten Anwendungsfälle ist aber ein Graph ausreichend. Wir ersparen uns damit die Einführung von Namen für die einzelnen Graphen. Das Zulassen mehrerer Graphen stellt aber kein prinzipielles Problem dar und ist eine einfache Erweiterung.

Darüberhinaus ist aber noch ein zweiter Graph wichtig, der ebenfalls im XRS-Modell enthalten ist. Für den Fall, dass Versionen aus grossen Datenmengen bestehen, kann es zur Speicherplatzeinsparung (Minimierung der redundant gespeicherten Information) interessant werden, einzelne Versionen nicht physisch voll zu speichern, sondern sie lediglich als Differenz zu einer anderen Version auszudrücken [KL 84, DLW 84]. Versionen können dann <u>vollständig</u> oder als <u>Differenzversion</u> gespeichert sein. Wird auf eine Differenzversion zugegriffen, müssen die nicht explizit gespeicherten Teile aus anderen Versionen bestimmt werden. Differenzversionen sind in drei Fällen interessant: Erstens wenn die Versionen gross sind, damit der zusätzliche Verwaltungsaufwand lohnt. Zweitens wenn die Versionen sich nicht wesentlich unterscheiden, damit die Differenzen nicht zu gross werden. Drittens wenn auf Differenzversionen relativ selten zugegriffen wird, da sie erst bestimmt werden müssen. Das XRS-Modell gestattet es zu bestimmen, ob eine Version vollständig oder als Differenz zu einer anderen Version derselben Versionenmenge gespeichert werden soll, wodurch wiederum gezielt Einfluss auf die Effizienz des Systems genommen werden kann [MP 86].

Jede Differenzversion bezieht sich auf genau eine vollständige oder eine Differenzversion, sie referenziert eine andere Version. Dadurch wird zwischen Versionen ein Graph definiert, den wir als Referenzgraphen bezeichnen. Neben den logischen Beziehungen des Präzedenzgraphen stellt der Referenzgraph physische Beziehungen innerhalb einer Versionenmenge dar.

Die Kanten eines Referenzgraphen sind von der referenzierenden zur referenzierten Version gerichtet. Der Referenzgraph einer Versionenmenge ist i.a. nicht zusammenhängend. Jede Differenzversion D hat genau eine Ausgangskante zu einer anderen Version; jede vollständige Version C hat keine Ausgangskante. Jede Version hat keine, eine oder mehrere Eingangskanten. Der Graph ist azyklisch.

Wir betrachten dazu Bild 3.3. Dargestellt ist eine Versionenmenge mit zehn Versionen, davon 4 vollständige Versionen. Jede Version repräsentiere beispielsweise die Konstruktion eines bestimmten Maschinenteils. Die Bedeutung des Präzedenzgraphen könnte etwa sein, dass die Nachfolgerversion aus der Vorgängerversion entstanden ist. Version 1 kann dann beispielsweise eine Archivversion sein, die zur Neukonstruktion des Teils (Version 4) herangezogen wurde. Die Versionen 2 und 3 stellen verworfene Konstruktionsversuche dar. Version 4 des Teils wird derzeit produziert und muss als vollständige Version schnell verfügbar sein. Diese Version wurde bereits als Grundlage für weitere Konstruktionsversuche (6 bis 10) genommen. Version 6 wird voraussichtlich dereinst Version 4 ersetzen. Das entsprechende Teil wird momentan in Einzelfertigung versuchsweise hergestellt, weshalb darauf häufig zugegriffen wird und es vollständig gespeichert ist [vgl. versioning strategies in DLW 84]. Die Versuche haben zu kleinen Änderungen Anlass gegeben, deren Konstruktion in Version 7 versucht wurde. Version 5 kann beispielsweise das gleiche Teil von der Konkurrenzfirma sein.

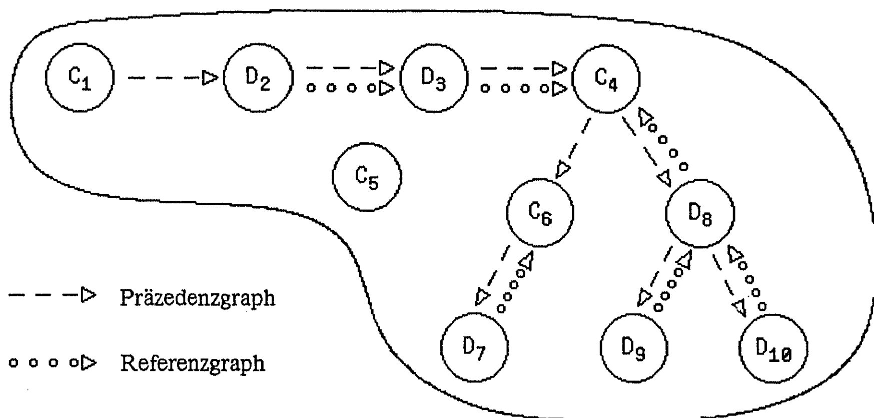


Bild 3.3: Versionenmenge mit Präzedenz- und Referenzgraph

4. Die Implementierung des XRS-Datenbankkerns

Nachdem wir im vorigen Kapitel die Konzepte des XRS-Datenbankkerns beschrieben haben, werden wir in diesem Kapitel die Struktur des Systems, die realisierende Schnittstelle und die Implementierung der Graphen und Differenzen vorstellen.

Im Vergleich mit konventionellen, relationalen Datenbanksystemen ersetzt die orzustellende Schnittstelle die Ein-Tupel-Schnittstelle [vgl. Hä 78]. Im XRS heisst die Kernschnittstelle eXtended Data System (XDS). Sie besteht aus einer Reihe von rozeduren, von denen die wichtigsten hier vorgestellt werden und deren Definitionen im nhang zu finden sind. XDS ist insofern eine Ein-Tupel-Schnittstelle, als dass mit einem rozeduraufruf genau ein Tupel eingefügt, verändert oder gelesen werden kann. Sie ist llerdings auch in vielfacher Hinsicht eine Mehr-Tupel-Schnittstelle. Gegenüber einer rein elationalen Ein-Tupel-Schnittstelle bietet sie ein stark erweitertes Navigier- und Cursorkonzept, wodurch einzelne Aufrufe nicht unabhängig voneinander sein müssen. usserdem wirken eine Reihe von Operatoren auf Tupelmengen. Lediglich der effektive Datenaustausch zwischen Anwenderprogramm und Datenbank beschränkt sich auf ein 'upel pro Aufruf.

.1 Der Systemaufbau

Bild 4.1 zeigt die Grobarchitektur des XRS-Systems. Der Memory Manager verwaltet einen ystempuffer zur Reduktion der Externspeicherzugriffe. Der Description Manager bietet eine hauptspeicherresidente Datenbankbeschreibung zum schnellen Zugriff auf Beschreibungsdaten an. Der Access Manager verwaltet Indexstrukturen mittels B*-Bäumen. Damit werden schnelle Zugriffspfade über beliebigen Attributen implementiert. Gitterdateien für M-Relationen werden durch den Gridfile Manager realisiert. Jedes Surrogat für das derzeit ein Tupel existiert, wird mit der zugehörigen Adresse des Tupels durch den Surrogate Manager verwaltet. Er ist auch für die Vergabe neuer Surrogate verantwortlich. Realisiert wird die Umsetzung der Surrogate auf Adressen über eine eigene Gitterdatei.

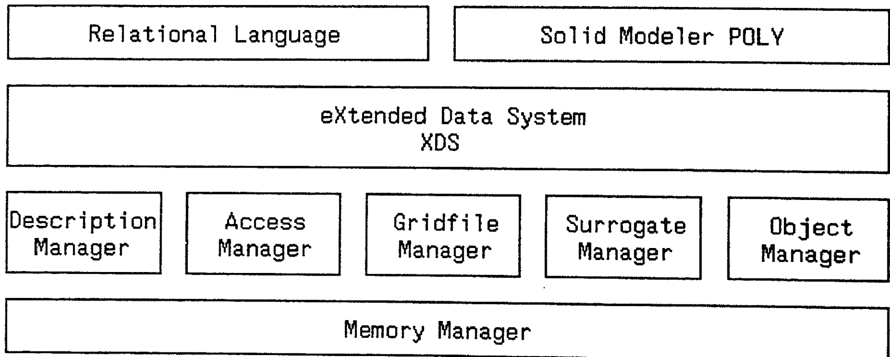


Bild 4.1: Grobarchitektur des XRS-Systems

Der Object Manager implementiert die Speicherungsstrukturen für K- und H-Relationen. K-Relationen stellen auf dieser Ebene einen Spezialfall von H-Relationen dar, weshalb der gleiche Speicherverwalter verwendet werden kann. Der Speicherverwalter speichert in jeder Hierarchieebene einer gegebenen, baumartigen Relationenhierarchie untergeordnete Tupel physisch unmittelbar hinter dem übergeordneten Tupel und clustert sie damit. Auf einer Datenseite stehen damit Tupel aus verschiedenen Relationen, jedoch mit fester Tupellänge einer jeden Relation. Untergeordnete Tupel können nicht ohne ihr übergeordnetes Tupel gespeichert werden. Damit wird das erste Clusterprinzip realisiert. Beim Löschen eines Tupels auf irgendeiner Ebene werden alle untergeordneten Tupel ebenfalls gelöscht. Bei allen Manipulationsoperationen sorgt bereits der Speicherverwalter für die Einhaltung der an V- und H-Referenzen gestellten Bedingungen. H-Referenzen werden mit dem übergeordneten Tupel abgeglichen und V-Referenzen mit dem eigenen Surrogat.

Die Schicht XDS ist für die Koordination der Leistungen der einzelnen Komponenten verantwortlich. Sie sorgt für die Einhaltung aller weiteren Clusterprinzipien, für eine konsistente Metadatenbank, die Wartung der Zugriffspfade, die Cursorverwaltung und ist nicht zuletzt für die Realisierung des Differenzverfahrens und der beiden Graphentypen zuständig. Die XDS-Schnittstelle ist derart konzipiert, dass sowohl eine deskriptive Sprachschicht, als auch ein Anwenderprogramm direkt darauf aufsetzen können.

Durch Einführung von Systemrelationen durch die Schicht XDS kann eine Clusterung der Versionen und Versionenmengen mit dem gleichen Speicherverwalter erreicht werden. So werden für jede K- und H-Relation zwei übergeordnete Relationen definiert. Pro Version wird in die untere der beiden ein Tupel eingetragen, alle Tupel der Version als darunter untergeordnete Tupel eingefügt und somit geclustert. Ebenso wird mit jeder Versionenmenge zur Clusterung aller enthaltenen Versionen verfahren.

XDS beinhaltet ein Metadatenbankkonzept zur Verwaltung der Datenbankbeschreibung. Der XDS-Benutzer definiert und verändert das Datenbankschema über genau die Prozeduren, die er auch zum Arbeiten mit der Datenbank verwendet. Das Metadatenbankschema ist eine H-Relation mit Relationen für die Objektklassen, Relationen, Attribute, Schlüssel und Attribut-Schlüssel-Paare. Der Benutzer kann jederzeit darin konsistente Änderungen vornehmen, wodurch beispielsweise neue Relationen deklariert oder bestehende samt Daten gelöscht werden können. Änderungen der Metadatenbank, die Reorganisationen nach sich ziehen würden, werden unterbunden.

Es muss erwähnt werden, dass es sich bei XRS um ein Pilotprojekt handelt und dem System deshalb auch einige Charakteristika von Datenbanksystemen abgehen. So existiert kein Transaktionskonzept und es ist ausschliesslich ein Einbenutzersystem.

4.2 Die XDS-Schnittstelle

Die XDS-Schnittstelle umfasst Prozeduren zur Datenmanipulation, zur Versionenverwaltung, zum Lesen aus der Datenbank, sowie einige administrative Prozeduren (Öffnen, ...), auf die hier nicht weiter eingegangen wird.

Als erste Manipulationsprozedur dient Insert dem Einfügen eines neuen Tupels in eine Relation. Der Benutzer spezifiziert die Relation, in die eingefügt werden soll. Will er eine neue Versionenmenge generieren, so genügt bereits diese Information. Spezifiziert er zusätzlich die Versionenmenge, aber keine Version, so wird in der Versionenmenge eine

neue Version erzeugt. Die Version gilt dann als vollständig gespeichert und mit keiner anderen Version in einem Graphen verbunden. Spezifiziert er schliesslich Relation, Versionenmenge und Version, so wird ein neues Tupel in die Version aufgenommen. Das Tupel muss im Format der entsprechenden Relation übergeben werden. Eventuell zu berücksichtigende H-Referenzen müssen im Tupel enthalten sein. Dem Benutzer wird das zugewiesene Surrogat des Tupels zurückgeliefert. Insert ist damit ein reiner Ein-Tupel-Operator. Die Semantik der einzelnen Surrogatteile kann hier ausgenutzt werden.

Unter Angabe eines vollständig spezifizierten Surrogats kann mit DeleteTuple ein Tupel aus einer Relation gelöscht werden. Hat das Tupel im Falle einer H-Relation abhängige Tupel, werden diese ebenfalls gelöscht. Hier hat also der Operator Auswirkungen auf mehrere Tupel.

Die Prozedur Replace dient dem Ändern von Attributwerten eines bereits vorhandenen Tupels. Das Surrogat des Tupels und die neuen Datenwerte müssen vollständig spezifiziert werden. Surrogat und H-Referenz dürfen nicht geändert werden. V-Referenzen werden auf ihre Korrektheit hin untersucht. Replace ist ein reiner Ein-Tupel-Operator.

All diese Operatoren sind jederzeit auf allen Tupeln bzw. Versionen anwendbar. Falls referenzierende Differenzversionen existieren, müssen deren physische Ausprägungen nachgeführt werden, damit sie sich logisch nicht mit der manipulierten Version ändern. Wir kommen damit zu den Prozeduren, die speziell der Versionenverwaltung dienen.

Mit DeleteVersion können alle Tupel einer Version und damit die ganze Version gelöscht werden. DeleteVersion ist also ein Mehr-Tupel-Operator. Wird die Version nicht spezifiziert, sondern nur die Objektklasse und Versionenmenge, so werden alle Versionen der Versionenmenge gelöscht. Entsprechendes gilt, falls nur die Objektklasse spezifiziert ist. Wird eine Version gelöscht, so muss sie aus den Graphen ausgekettet werden. Wird eine vollständige Version gelöscht, werden alle referenzierenden Versionen vervollständigt, d.h. in vollständige Versionen gewandelt. Ist die zu löschende Version eine Differenzversion, werden die referenzierenden Versionen geändert, so dass sie sich auf die referenzierte Version beziehen. Im Präzedenzgraphen wird jeder Vorgänger mit jedem Nachfolger verbunden. Würde in unserem Beispiel aus dem vorigen Kapitel Version 8 gelöscht, so würden anschliessend die Versionen 9 und 10 Version 4 referenzieren und Version 4 würde zum direkten Vorgänger von 9 und 10.

Mit Copy kann eine Version innerhalb einer Versionenmenge kopiert werden. Soll beispielsweise Version 4 verfügbar bleiben, daran aber weitere Konstruktionsvarianten versucht werden, so wird Version 4 kopiert und an den kopierten Versionen Änderungen vorgenommen. Mit dem Operator spezifiziert der Benutzer die Version sowie die Einbindung der beiden beteiligten Versionen in die Graphen. Er kann dabei wählen, ob die kopierte Version vollständig oder eine der beiden als Differenz zur anderen gespeichert werden soll. Ebenso wird angegeben, ob eine Vorgänger-/Nachfolgerbeziehung besteht und gegebenfalls wie herum. Zurückgegeben wird dem Benutzer die Identifikation der neuen Version. Unabhängig von der physischen Repräsentation wird für alle Tupel der neuen Version ein neues Surrogat vergeben und vom Surrogate Manager verwaltet. Die Struktur des Surrogats erlaubt dies durch einfachen Austausch der Versionennummer. Auf die gleiche Art werden in einer kopierten Version H- und V-Referenzen aktualisiert damit sie korrekt innerhalb der Version referenzieren. Im Beispiel ist Version 6 als vollständiger Nachfolger aus Version 4 entstanden. Version 7 ist als Differenz zum Vorgänger entstanden, während Version 4 als vollständiger Nachfolger aus 3 entstanden ist. Version 3 wurde dabei als Differenz zur neuen Version ausgedrückt.

Mit Complete kann eine Differenzversion in eine vollständige Version gewandelt werden. Dies ist notwendig, wenn sich nachträglich herausstellt, dass auf eine Version sehr häufig zugegriffen werden soll. Als Folge des Operators wird der Referenzgraph entsprechend geändert, während der Präzedenzgraph unverändert bleibt. Sollte sich herausstellen, dass Version 7 sehr häufig benötigt wird, ist es sinnvoll sie zu vervollständigen, womit die Kante im Referenzgraphen entfällt.

Die Prozedur Invert dient dem Invertieren der physischen Darstellung eines Paares einer vollständigen und einer sie referenzierenden Differenzversion. Die Differenzversion wird zur vollständigen Version und die vollständige Version zur referenzierenden Differenzversion. Durch wiederholte Anwendung des Operators kann eine ganze Kette mit einer vollständigen und n Differenzversionen sozusagen herumgedreht werden. Der Operator wirkt sich nicht auf den Präzedenzgraphen aus.

Es muss betont werden, dass die letzten vier Operatoren sehr mächtig sind, werden doch damit ganze Versionen manipuliert.

Mit den beiden Prozeduren Connect und Disconnect kann ausschliesslich der Präzedenzgraph verändert werden. Zwischen zwei spezifizierten Versionen derselben Versionenmenge kann eine logische Beziehung hergestellt bzw. entfernt werden.

Navigate erlaubt das Navigieren auf einem zusammenhängenden Teil des Präzedenz- und Referenzgraphen einer Versionenmenge. Der Benutzer spezifiziert eine Ausgangsversion, den Graphen und die Richtung in die er sich bewegen will. Er erhält dann die Identifikation, beispielsweise einer Nachfolgerversion zurück. Durch wiederholten Aufruf kann er dann unter Angabe der zuletzt erhaltenen Version als "Cursor-Version" alle Nachfolger erhalten.

Zwei Retrieve-Operatoren dienen dem Wiedergewinnen gespeicherter Information. Beides sind Ein-Tupel-Operatoren insofern, als dass bei einem Aufruf ein einziges Tupel in das Anwenderprogramm übertragen wird. Dennoch hat der Benutzer eine mengenorientierte Sicht: Mit jedem Aufruf wird eine Menge von Tupeln identifiziert. Es können nacheinander alle Tupel der Menge gelesen werden. Bei einer klassischen Ein-Tupel-Schnittstelle ist die Menge immer eine Relation, deren einzelne Tupel nacheinander gelesen werden. An der XDS-Schnittstelle können alle Versionenmengen einer Relation, alle Versionen einer Versionenmenge oder alle Tupel einer Version gelesen werden. Der Benutzer kann sich dabei auf eine Relation beschränken, indem er den Operator RetrieveRelational verwendet, oder er kann relationenübergreifend die Datenbank objektorientiert sehen, indem er mit RetrieveObjectOriented arbeitet. Mit dem zweiten Operator kann er insbesondere alle Tupel einer Version lesen.

Mit einem Retrieve-Aufruf spezifiziert der Benutzer ein Ordnungskriterium für die ausgewählte Menge. Die Ordnung von Tupeln kann systemdefiniert oder durch einen Schlüssel gegeben sein, wobei Schlüssel entweder durch Indexstrukturen permanent unterstützt sind (B*-Bäume), oder zum Anfragezeitpunkt eine sortierte Liste der Tupel erstellt werden muss. Über einen weiteren Parameter steuert der Benutzer die Richtung in der gemäss der Ordnung navigiert werden soll.

Bei jedem Retrieve-Aufruf werden alle Parameter voll spezifiziert. Das Surrogat des zuletzt erhaltenen Tupels muss als Cursor mitgeliefert werden. Dadurch ist es sowohl möglich, intern weitere Cursorinformation zur Beschleunigung der Anfrage bereitzuhalten, als auch zwischenzeitliche Änderungen an der Menge (Löschen, Einfügen eines Tupels, ...) sofort berücksichtigen zu können. Der Benutzer kann damit auch an jeder beliebigen Stelle mit

einem Surrogat direkt aufsetzen und das zugehörige Tupel oder ein "Nachbartupel" lesen.

4.3 Implementierung der Graphen und Differenzen

Um den Präzedenzgraphen zu speichern, müssen wir minimal zu jeder Version ihre Vorgänger- oder Nachfolgerversionen ablegen. Um gleichmässig schnell auf der Versionenmenge navigieren zu können, speichern wir zu jeder Version sowohl einen Verweis auf die Vorgänger wie einen auf die Nachfolger. Es wurde erwähnt, dass zur Clusterung aller Tupel einer Version in eine Systemrelation je ein Tupel pro Version eingetragen wird. Genau in einem solchen Tupel lässt sich auch der Präzedenzgraph speichern, und zwar unter Verwendung des vorhandenen Speicherverwalters. (Falls die Objektklasse nur aus M-Relationen besteht, muss eine eigene Systemrelation generiert werden.) Wir beschränken uns bei der Implementierung auf eine Maximalzahl von n Vorgängern und m Nachfolgern pro Version und generieren in der Systemrelation n + m Attribute zum Speichern je einer Versionennummer. Die Operatoren, die eine Kante im Präzedenzgraphen löschen oder setzen, verändern dann genau die beiden Tupel der beiden beteiligten Versionen.

Ebenso lässt sich der reine Referenzgraph ablegen, wobei insofern eine Vereinfachung auftritt, als dass eine Version höchstens eine Referenzversion hat. Zur Speicherung einer Differenzversion, die identisch ihrer Referenzversion ist, genügt dies bereits. Schwieriger wird es, wenn Differenz- oder Referenzversion geändert werden.

Ein Tupel wird in eine vollständige Version eingetragen, daraus gelöscht oder darin geändert, indem es physisch eingetragen, gelöscht oder geändert wird. Wird eine Differenzversion manipuliert, wird in allen drei Fällen das Tupel physisch aufgenommen mit einem Vermerk, um welche Form der Änderung es sich handelt. In allen referenzierenden Differenzversionen wird dann das Tupel physisch eingetragen, und zwar markiert als in der Version gelöscht, hinzugefügt oder geändert. Eine detailliertere Darstellung findet sich in [MP 86]. Das Differenzverfahren hat als Einheit der Differenzbildung einzelne Tupel. Eine Differenzversion enthält also alle Tupel, in denen sich die Version von der Referenzversion unterscheidet. Damit kann wieder der gleiche Speicherverwalter auch für Differenzen verwendet werden. Lediglich in H-Tupeln hat dies zur Folge, dass zu einem geänderten, abhängigen Tupel alle seine übergeordneten, nicht geänderten Tupel auch gespeichert sein müssen. Um insbesondere die Retrieve-Operatoren effizient zu implementieren, wird zusätzlich eine Zuordnung zwischen Differenz- und Referenzversion auf Tupelebene gespeichert. So enthält jedes Tupel einen Verweis (Versionen- und Laufnummer) auf sein Referenztupel und alle referenzierenden Tupel. Eine Maximalzahl referenzierender Tupel erlaubt wieder feste Tupellänge.

5. Schlussbemerkungen

In der Literatur bezüglich der bei neuen Anwendungen zu lösenden Probleme findet sich auch das Problem, dass von einem technischen Objekt mehrere Alternativen modelliert und gespeichert werden müssen. Nach unserem Verständnis des Begriffs stellen Alternativen spezielle Versionen dar. In einer Konstruktionsumgebung sind zwei alternative Entwürfe zwei Versionen, die aus einer gemeinsamen Version hervorgegangen sind. Ein Gegenstand, der in mehreren Alternativen vorliegt, ist in der Datenbank durch mehrere Versionen

repräsentiert, die in ihren Ausprägungen weitgehend gleich sind. In beiden Fällen unterstützt der hier vorgestellte Datenbankkern die Implementierung der gewünschten Semantik in einer höheren Schicht, im ersten Fall durch die Graphverwaltung, im zweiten Fall durch die Differenzbildung.

Ebenfalls häufig wird die Modellierung von Varianten gefordert. In der Maschinenkonstruktion stellen Varianten parametrisierte Konstruktionen dar. Der Konstruktionsprozess ist für zwei Varianten der gleiche, aber mit verschiedenen Parametern. Betrachten wir die zu speichernden Daten, so werden in unserem Modell zwei Varianten auch durch zwei Versionen repräsentiert. Wird in der Version der Konstruktionsprozess gespeichert (z.B. CSG-Ansatz), so unterscheiden sich die Versionen wieder nur in einigen Datenwerten. Wird die fertige Konstruktion gespeichert (z.B. Randdarstellung), kann die Topologie der Varianten identisch sein und Unterschiede können nur in der Geometrie auftreten. Im Beispiel aus Kapitel 3 (Bild 3.2) treten dann nur in der Relation `vertex` Unterschiede zwischen zwei Varianten auf. Für beide Fälle bietet XRS Unterstützung bei der physischen Verwaltung.

Die Datenbankentwickler werden vor allem aus dem VLSI-Bereich gedrängt, in ihre Modelle den Begriff der Repräsentation zu integrieren. Zu verstehen ist darunter, dass ein Objekt in mehreren Darstellungsformen beschrieben sein kann. Das Problem besteht dann vor allem darin, die verschiedenen Darstellungsformen untereinander konsistent zu halten. In unserem Beispiel sind dies die CSG- und die Randdarstellung. Wir schlagen die Aufnahme aller Repräsentationen in eine Objektklasse vor (siehe Bild 3.2). Die Operationen, die sich auf Versionen beziehen, wirken dann gleichermassen auf alle Repräsentationen, was ein erster Schritt zur Konsistenz ist. Unsere Implementierung optimiert dann auch die Speicherung und den Zugriff. Die vollständige Konsistenzprüfung muss freilich auf höherer Ebene gemacht werden.

Für viele Anwendungen spielt das Zuordnen von Eigenschaften zu einzelnen Versionen eine grosse Rolle [DL 85]. Wir haben in den Datenbankkern keine derartigen Konzepte aufgenommen, weil die Speicherung der Eigenschaften in normalen Attributen möglich ist und die zugehörige Semantik anwendungsabhängig ist. Aus den gleichen Gründen wurde zwar ein Datentyp TIMEPOINT integriert, aber auf die explizite Navigiermöglichkeit entlang einer Zeitachse verzichtet. Um eine zeitlich geordnete Folge von Versionen zu erzeugen, definiert der Benutzer einen Schlüssel über einem Zeitattribut und liest die Versionen nach auf- oder absteigenden Schlüsselwerten.

Insgesamt haben wir in diesem Aufsatz einen anwendungsneutralen Datenbankkern vorgestellt. Wir haben gezeigt, wie unter Erweiterung des Relationenmodells die Modellierung beliebiger Objekte möglich ist. Durch stabile Surrogate mit Struktur und spezielle Referenz-Attributtypen wird die Objekt- und Versionenzusammengehörigkeit mehrerer Tupel dargestellt. Verschiedene Relationentypen ermöglichen stärkere semantische Integrität und verbessern das Leistungsverhalten einer Implementierung. Graphen auf Versionen dienen der Darstellung physischer und logischer Abhängigkeiten zwischen den Versionen.

Wesentliche Punkte der Implementierung des Datenbankkerns von XRS wurden vorgestellt. Verschiedene Clusterprinzipien minimieren den Seitentransfer zwischen Haupt- und Sekundärspeicher. Eine Reihe von Operationen realisieren eine anwendungsneutrale Versionenverwaltung. Die physische Repräsentation der Versionen lässt sich durch Differenzenbildung optimieren. Wir hoffen, mit diesen Konzepten und deren Realisierung einen Beitrag zu einem leistungsstarken Datenbankkern liefern zu können.

Literaturverzeichnis

[BP 85] Blaser, A., Pistor, P. (Hrsg.): Datenbank-Systeme für Büro, Technik und Wissenschaft. Informatik-Fachberichte Nr. 94, Springer-Verlag, Berlin, 1985.

[DL 85] Dittrich, K. R., Lorie, R. A.: Version Support for Engineering Database Systems. IBM Research Report RJ 4769, San Jose, 1985, pp. 1-19.

[DLW 84] Dadam, P., Lum, V., Werner, H.-D.: Integration of Time Versions into a Relational Database System. in: Proc. 10th Int. Conference on VLDB, Singapore, 1984, pp. 509 - 522.

[Hä 78] Härder, T.: Implementierung von Datenbanksystemen. Carl Hanser Verlag, München, 1978.

[Hä 84] Härder, T.: Überlegungen zur Modellierung und Integration der Zeit in temporalen Datenbanksystemen. Bericht Nr. 19/84, Universität Kaiserslautern, Fachbereich Informatik, 1984, S. 1-30.

[Hi 85] Hinrichs, K.: Implementation of the GRID File: Design Concepts and Experience. BIT 25, 1985, pp. 569-592.

HOT 76] Hall, P., Owlett, J., Todd, S.: Relations and Entities. in: Modelling in Data Base Management Systems, G. M. Nijssen (ed.), North-Holland Publishing Company, 1976, pp. 201 - 220.

KL 84] Katz, R. H., Lehman, T. J.: Database Support for Versions and Alternatives of Large Design Files. in: IEEE Transactions on Software Engineering, Vol. SE-10, No. 2, March 1984, pp. 191-200.

[KSW 86] Klahold, P., Schlageter, G., Wilkes, W.: A General Model for Version Management in Databases. Informatik Berichte Nr. 58, Fern-Universität Hagen, 1986, pp. 1-22.

[LP 83] Lorie, R., Plouffe, W.: Complex Objects and Their Use in Design Transactions. in: Proc. Annual Meeting on Databases for Design Applications, San Jose, CA, USA, May 1983, pp. 115-121.

[Mei 86] Meier, A.: Applying Relational Database Techniques to Solid Modelling. in: Computer-aided design, Vol. 18, No. 6, July/August 1986, pp. 319-326.

[Mel 86] Meloni, T.: Kopplung des Modellierers POLY mit dem Datenbankkern von XRS. Diplomarbeit, Institut für Informatik, ETH Zürich, Sept. 1986.

[Mi 84] Mitschang, B.: Überlegungen zur Architektur von Datenbanksystemen für Ingenieuranwendungen. in: Proc. 14. GI-Jahrestagung, Braunschweig, Informatik-Fachberichte Nr. 88, Springer-Verlag, Berlin, 1984, S. 318-334.

[Mi 85] Mitschang, B.: Charakteristiken des Komplex-Objekt-Begriffs und Ansätze zu dessen Realisierung. in: [BP 85], S. 382-400.

[MP 86] Meier, A., Petry, E.: Versionenkontrolle geometrischer Daten. in: Proc. 16. GI-Jahrestagung, Berlin, Okt. 1986.

[NHS 84] Nievergelt, J., Hinterberger, H., Sevcik, K.: The Grid File: An Adaptable, Symmetric Multikey File Structure. ACM Transactions on Database Systems, Vol. 9, No. 1, March 1984, pp. 38-71.

[Po 86] Poretti, G.: Integration von SURROGATE, TENSOR, SET und SEQUENCE in einer Datenbankprogrammiersprache. Diplomarbeit, Institut für Informatik, ETH Zürich, Aug. 1986.

[POLY 86] Meier, A., Loacker, H.-B., Paquet, F., Kohler, T.: Das rechnergestützte Unterrichtssystem POLY zur Darstellung und Manipulation ebenbegrenzter Objekte. GI-Tagung "Informatik Grundbildung in Schule und Beruf", Universität Kaiserslautern, September 1986.

[Re 80] Requicha, A. A. G.: Representations for Rigid Solids: Theory, Methods and Systems. in: Computing Surveys, Vol. 12, No. 4, Dec. 1980, pp. 437-464.

[SS 77] Smith, J. M., Smith, D. C. P.: Database Abstractions: Aggregation and Generalization. in: ACM Transactions on Database Systems, Vol. 2, No. 2, June 1977, pp. 105-133.

[XRS 87] Meier, A., Durrer, K., Heiser, G., Petry, E., Wälchli, A., Zehnder, C. A.: XRS: Ein erweitertes relationales Datenbanksystem zur Verwaltung von technischen Objekten und Versionen. in: Bericht Nr. 76, Institut für Informatik, ETH Zürich, 1987 (in diesem Band).

Anhang: Die XDS-Schnittstelle

Die hier beschriebene prozedurale XDS-Schnittstelle unterscheidet sich von der in [XRS 87] vorgestellten Schnittstelle. Die wesentlichsten Unterschiede bestehen in der Einführung von Operationen auf ganzen Versionen und den Graphen.

Vor der Definition der Operationen der XDS-Schnittstelle sind im folgenden die zum Verständnis wichtigen Typdefinitionen der Schnittstelle aufgeführt. Darin ist ebenfalls die Definition des Metadatenbankschemas enthalten. Diejenigen Prozeduren, die nicht direkt der Datenmanipulation und Abfrage dienen, sind weggelassen.

```
TYPE
(* Important types for XDS-interface: *)
     SurBaseType = CARDINAL;
     Surrogate   = RECORD
                      relNo: SurBaseType;
                      setNo: SurBaseType;
                      verNo: SurBaseType;
                      seqNo: SurBaseType
                    END;
     TupleId     = Surrogate;
     VersionId   = RECORD
                      objectClass: SurBaseType;
                      objectSet  : SurBaseType;
                      version    : SurBaseType
                    END;
     HReference  = Surrogate;
     VReference  = Surrogate;
     Reference   = Surrogate;
     Tuple       = ADDRESS;
     TimePoint   = RECORD
                      year  : INTEGER;
                      month : [1 .. 12];
                      day   : [1 .. 31];
                      hour  : [0 .. 23];
                      minute: [0 .. 59];
                      second: [0 .. 59]
                    END;
     PhysicalRepresentation = (complete, differenceFrom, differenceTo);
     LogicalPrecedence      = (noPrecedence, predecessorFrom, predecessorTo);
     Granule                = (set, version, tuple);
     Direction              = (first, next, current, previous, last,
                               subordinate, superordinate);
     Order                  = (key, system);
     NavigationDirection    = (anyPredecessor, nextPredecessor,
                               anyDescendant, nextDescendant,
                               anyReferencing, nextReferencing, referenced);
     ReturnInfo             = (errors, success, none, relationChange);
```

```
(* Metadatabasescheme definition: *)
   ObjClsDes    = RECORD
                    sur         : Surrogate;
                    number      : SurBaseType;
                    name        : NameType
                  END;
   RelDes       = RECORD
                    sur         : Surrogate;
                    objCls      : HReference;
                    name        : NameType;
                    number      : SurBaseType;
                    type        : RelType;
                    father      : Reference
                  END;
   AttDes       = RECORD
                    sur         : Surrogate;
                    relation    : HReference;
                    name        : NameType;
                    type        : AttType;
                    minAttValue,
                    maxAttValue: AttValue
                  END;
   KeyDes       = RECORD
                    sur         : Surrogate;
                    relation    : HReference;
                    name        : NameType;
                    keyType     : KeyType
                  END;
   AttKeyDes    = RECORD
                    sur         : Surrogate;
                    key         : HReference;
                    attribute   : VReference;
                    seqNoInKey  : CARDINAL
                  END;
```

```
PROCEDURE Insert (tupleId  : TupleId;
                  tupleData: Tuple;
                  VAR sur  : Surrogate;
                  VAR info : ReturnInfo);
```

Mit der Prozedur `Insert` werden neue Tupel in die Datenbank eingefügt. Parameter `tupleId` enthält die Information, in welche Relation, Versionenmenge und Version das Tupel eingefügt werden soll. `tupleData` ist ein Pointer zum einzufügenden Tupel. Das generierte Surrogat des Tupels wird im Parameter `sur` zurückgereicht.

Parameter **`tupleId`** muss in jedem Fall teilspezifiziert sein. `tupleId.relNo` spezifiziert immer die Relation, in die das Tupel einzufügen ist.

Falls `tupleId.setNo` spezifiziert ist, wird das Tupel in diese Versionenmenge aufgenommen. Ist der Parameter unspezifiziert, wird innerhalb der Objektklasse, zu der die Relation gehört, eine neue Versionenmenge generiert und das Tupel als erstes Element darin aufgenommen. Automatisch wird damit auch eine erste Version in der Versionenmenge erzeugt, der das Tupel angehört. Dies ist der Mechanismus,

mit dem Versionenmengen und deren erste Versionen in die Datenbank eingebracht werden.

Ist `tupleId.setNo` unspezifiziert, so wird `tupleId.verNo` ignoriert. Ist `tupleId.setNo` dagegen spezifiziert, kann `tupleId.verNo` die Nummer einer Version angeben, in die das Tupel einzufügen ist. Falls keine solche Version bezeichnet ist, wird eine neue Version in die Versionenmenge aufgenommen. Das Tupel wird deren erstes Element. Auf diese Weise können neue Versionen erzeugt werden. Eine neue Version ist als vollständige Version gespeichert und in keinem Graphen mit einer anderen Version verbunden.

`tupleId.seqNo` muss immer unspezifiziert übergeben werden. Diese Laufnummer innerhalb der Relation, Versionenmenge und Version wird vom System generiert.

tupleData ist die Adresse des einzufügenden Tupels. Die einzelnen Attribute werden, wie im Datenbankschema spezifiziert, erwartet. Im Falle des Einfügens in eine abhängige Relation bestimmt der Wert des H-Referenz-Attributs, an welches übergeordnete Tupel das Tupel anzugliedern ist. Dieses Tupel muss logisch und physisch existieren. Existiert es nur logisch, wird es automatisch physisch eingefügt. Entsprechendes gilt für indirekt übergeordnete Tupel. V-Referenz-Attribute im einzufügenden Tupel dürfen entweder unspezifiziert sein, oder sie müssen ein Tupel innerhalb der Version referenzieren. Beim Einfügen in eine M-Relation bestimmt der im Tupel enthaltene mehrdimensionale Schlüssel die physische Position des Tupels auf dem Sekundärspeicher. Das Surrogat-Attribut muss beim Aufruf im Tupel keinen Wert enthalten. Dieser wird bei erfolgreicher Ausführung der Prozedur vom System eingefügt.

Das Datenbankschema wird ebenfalls mittels der Prozedur `Insert` definiert, indem Einfügungen in die vordefinierten Relationen des Metadatenbankschemas gemacht werden. Dabei wird die semantische Korrektheit von Schemadefinitionen überwacht. Versionen des Datenbankschemas werden nicht unterstützt, weshalb im Parameter `tupleId` lediglich die Relation zu spezifizieren ist. Es dürfen keine Einfügungen in Objektklassendefinitionen vorgenommen werden, zu denen bereits Daten existieren.

Parameter **sur** enthält nach erfolgreicher Ausführung der Prozedur das voll spezifizierte Surrogat des eingefügten Tupels.

Über das erfolgreiche Einfügen des Tupels informiert der Returnparameter **info**. Ist ein Fehler aufgetreten, kann eine genaue Fehlermeldung durch Aufruf der speziellen Operation `CheckDB` erhalten werden.

```
PROCEDURE DeleteTuple (tupleId : TupleId;
                       VAR info: ReturnInfo);
```

Mit `DeleteTuple` werden Tupel aus der Datenbank entfernt. Parameter `tupleId` muss das Surrogat des zu löschenden Tupels enthalten.

tupleId muss in allen Teilen voll spezifiziert sein. Gehören abhängige Tupel zu dem zu löschenden Tupel, so werden diese ebenfalls gelöscht. Dies gilt ebenfalls für indirekt abhängige Tupel. Falls das Tupel zu einer Version gehört, die referenzierende Versionen hat, werden eventuelle referenzierende Tupel physisch in die referenzierenden Versionen eingefügt, damit diese sich logisch nicht verändern.

Die Operation ist ebenfalls auf die Metadatenbank anwendbar. Konsistenzverletzende Löschungen werden unterbunden. Es dürfen nur (Teile von) Objektklassendefinitionen gelöscht werden, die noch keine Daten enthalten.

Returnparameter **info** informiert darüber, ob Daten gelöscht wurden, ob das Tupel gefunden wurde, oder ob die Operation fehlerhaft abgeschlossen wurde. Im letzten Fall kann `CheckDB` zur genaueren Diagnose aufgerufen werden.

```
PROCEDURE Replace (tupleId  : TupleId;
                   tupleData: Tuple;
                   VAR info : ReturnInfo);
```

Mit `Replace` können die Attributwerte eines Tupels verändert werden. `tupleId` muss das Tupel eindeutig bezeichnen. `tupleData` wird als Pointer zum Tupel mit den neuen Datenwerten interpretiert.

Alle Teile von **tupleId** müssen spezifiziert sein. `tupleId` ist das Surrogat des zu ändernden Tupels. Gehört das Tupel zu einer Version, die referenzierende Versionen hat, so werden eventuell vorhandene referenzierende Tupel, die nur logisch existieren, physisch eingefügt und damit von dem geänderten Tupel unabhängig.

Die Attributwerte des neuen Tupels werden im Parameter **tupleData**, wie im Datenbankschema definiert, erwartet. Die Attributwerte des Surrogats und einer eventuell vorhandenen H-Referenz können nicht geändert werden. V-Referenzen können entweder unspezifiziert sein, oder sie müssen ein Tupel innerhalb der Version referenzieren. Die Existenz des referenzierten Tupels wird nicht geprüft.

`Replace` ist ebenfalls auf die Metadatenbank anwendbar, sofern keine Konsistenzbedingungen verletzt werden und das Tupel zu einer Objektklassendefinition gehört, zu der es noch keine Daten gibt.

Der Returnparameter **info** enthält die Information, ob ein Tupel erfolgreich geändert wurde, ob kein Tupel gefunden wurde oder ob ein Fehler aufgetreten ist. Eventuell können mit `CheckDB` Fehlermeldungen abgefragt werden.

```
PROCEDURE DeleteVersion (version : VersionId;
                         VAR info: ReturnInfo);
```

Ganze Versionen, Versionenmengen oder Objektklassen können mit `DeleteVersion` gelöscht werden Parameter `version` spezifiziert die zu löschenden Daten.

Parameter **version** kann teilspezifiziert sein. Die Objektklasse, in der zu löschen ist, muss immer in `version.objectClass` bezeichnet sein.

Falls `version.objectSet` unspezifiziert ist, wird der Inhalt der gesamten Objektklasse gelöscht. Dies betrifft alle Versionenmengen und deren Versionen innerhalb der Objektklasse.

Ist `version.objectSet` unspezifiziert, bleibt `version.version` unberücksichtigt. Andernfalls kann `version.version` unspezifiziert sein, wodurch alle Versionen der Versionenmenge gelöscht werden. Ist `version.version` spezifiziert, wird damit genau eine zu löschende Version bezeichnet. In diesem Fall wird die Version gelöscht, indem alle zur Version gehörenden Tupel gelöscht werden.

Beim Löschen einer einzelnen Version sind andere Versionen der gleichen Versionenmenge in folgender Weise über mögliche Graphverbindungen betroffen: Die `version` referenzierenden Versionen dürfen sich logisch nicht verändern. Ist `version` eine vollständige Version, so werden deshalb alle referenzierenden Versionen physisch vervollständigt. Ist `version` dagegen eine Differenzversion, werden die referenzierenden Versionen in ihrer physischen Repräsentation derart verändert, dass sie sich anschliessend auf diejenige Version beziehen, die vorher von `version` referenziert wurde. Der Referenzgraph wird entsprechend gewartet. Im Präzedenzgraphen werden alle Vorgängerversionen von `version` zu direkten Vorgängern aller Nachfolgerversionen von `version`.

Die Operation ist nicht auf die Metadatenbank anwendbar, da keine Versionen des Datenbankschemas unterstützt werden.

Parameter **info** informiert, ob Versionen gelöscht wurden, ob solche nicht gefunden wurden, oder ob die Operation fehlerhaft abgelaufen ist. Fehlermeldungen können mittels `CheckDB` abgefragt werden.

```
PROCEDURE Copy (fromVersion              : VersionId;
                physicalRepresentation: PhysicalRepresentation;
                logicalPrecedence        : LogicalPrecedence;
                VAR toVersion            : VersionId;
                VAR info                 : ReturnInfo);
```

Mit Copy kann die Version fromVersion innerhalb ihrer Versionenmenge kopiert werden. Dabei hat man über den Parameter physicalRepresentation die Möglichkeit, die physische Darstellung der alten und der kopierten Version zu bestimmen. Ihre logische Verbindung zueinander wird über logicalPrecedence spezifiziert. Der generierte Bezeichner der kopierten Version wird im Parameter toVersion zurückgereicht.

Parameter **fromVersion** muss in allen drei Teilen spezifiziert sein und bezeichnet damit eine Version, die zu kopieren ist.

Über den Parameter **physicalRepresentation** nimmt der Benutzer Einfluss auf die physische Darstellung der zu kopierenden und der kopierten Version: Der Wert complete besagt, dass die kopierte Version als vollständige Version gespeichert werden soll, sie also physisch alle sie umfassenden Tupel enthalten soll. Die zu kopierende Version wird in ihrer Darstellung nicht tangiert. Mit differenceFrom wird angegeben, dass die Ausgangsversion fromVersion als Differenz zur kopierten Version repräsentiert werden soll. Dies ist nur möglich, falls fromVersion beim Aufruf der Prozedur eine vollständige Version ist. Hat physicalRepresentation den Wert differenceTo, so ist umgekehrt die kopierte Version als Differenz zur Ausgangsversion zu speichern.

Die Beziehung im Präzedenzgraphen zwischen den beiden Versionen kann direkt über den Parameter **logicalPrecedence** gewählt werden. Der Wert noPrecedence besagt, dass zu kopierende und kopierte Version unabhängig voneinander sein sollen. predecessorFrom macht die zu kopierende Version zu einem direkten Vorgänger der kopierten Version. Umgekehrt wird durch predecessorTo die zu kopierende Version zu einem direkten Nachfolger der kopierten Version.

Die Operation ist nicht auf die Metadatenbank anwendbar.

Die Identifikation der generierten Version wird vom System vergeben und im Parameter **toVersion** zurückgereicht, falls die Operation erfolgreich durchgeführt wurde.

Über die Ausführung der Operation informiert der Parameter **info**. Eventuelle Fehlermeldungen können über Aufrufe von CheckDB erhalten werden.

```
PROCEDURE Complete (version : VersionId;
                    VAR info: ReturnInfo);
```

Mit Complete kann eine Version vervollständigt werden. Die Version muss mit dem voll spezifizierten Parameter version bezeichnet werden.

Die mit **version** identifizierte Version ist nach erfolgreicher Ausführung der Operation als vollständige Version gespeichert. Falls sie bereits bei Aufruf der Prozedur eine vollständige Version ist, hat die Operation keine Auswirkung. Ist sie dagegen eine Differenzversion, so werden alle Tupel, die die Version ausmachen, physisch gespeichert. Gegebenenfalls wird der Referenzgraph entsprechend angepasst. Der Präzedenzgraph ist nicht berührt.

Die Operation ist nicht auf die Metadatenbank anwendbar.

Parameter **info** signalisiert, ob ein Fehler aufgetreten ist, oder ob die Operation korrekt ausgeführt wurde. Falls ein Fehler aufgetreten ist, kann eine genaue Fehlermeldung durch Aufruf der speziellen Operation CheckDB erhalten werden.

```
PROCEDURE Invert (differenceVersion,
                  completeVersion    : VersionId;
                  VAR info           : ReturnInfo);
```

`Invert` vertauscht die physische Repräsentation einer Differenzversion `differenceVersion` und einer vollständigen Version `completeVersion`.

Mit Parameter **`differenceVersion`** wird eine Differenzversion identifiziert, die die vollständige Version **`completeVersion`** referenzieren muss. Die Operation ändert die physische Repräsentation der Versionen in der Weise, dass die ursprüngliche Differenzversion in eine vollständige Version gewandelt wird, und die ursprünglich vollständige Version als Differenzversion zu dieser repräsentiert wird. Der Referenzgraph wird entsprechend angepasst, während der Präzedenzgraph nicht verändert wird.

Die Operation ist nicht auf die Metadatenbank anwendbar.

Returnparameter **`info`** signalisiert, ob die Operation erfolgreich war. Sind Fehler aufgetreten, können über `CheckDB` Fehlermeldungen abgefragt werden.

```
PROCEDURE Connect (fromVersion,
                   toVersion    : VersionId;
                   VAR info     : ReturnInfo);
```

Mit `Connect` wird eine logische Verbindung zwischen zwei Versionen `fromVersion` und `toVersion` derselben Versionenmenge etabliert.

`fromVersion` und **`toVersion`** müssen zwei existierende Versionen einer Versionenmenge identifizieren. Beide Versionen müssen nicht verschieden sein. `fromVersion` wird mit der Operation zur Vorgängerversion von `toVersion` gemacht. Existiert die Beziehung zwischen den Versionen bereits, hat die Operation keine Auswirkungen auf den Präzedenzgraphen.

`Connect` ist nicht auf die Metadatenbank anwendbar. Der Referenzgraph ist nicht berührt.

Über eventuell aufgetretene Fehler informiert der Returnparameter **`info`**. Fehlermeldungen kann man mit `CheckDB` abrufen.

```
PROCEDURE Disconnect (fromVersion,
                      toVersion    : VersionId;
                      VAR info     : ReturnInfo);
```

Eine Verbindung des Präzedenzgraphen zwischen den zwei Versionen `fromVersion` und `toVersion` kann mit `Disconnect` entfernt werden.

`fromVersion` und **`toVersion`** müssen zwei existierende Versionen einer Versionenmenge identifizieren. Beide Versionen müssen nicht verschieden sein. Falls `fromVersion` eine direkte Vorgängerversion von `toVersion` ist, wird diese Beziehung aus dem Präzedenzgraphen entfernt.

`Disconnect` ist nicht auf die Metadatenbank anwendbar. Der Referenzgraph ist nicht berührt.

Returnparameter **`info`** informiert darüber, ob die Operation korrekt oder fehlerhaft abgeschlossen wurde. Im letzteren Fall kann `CheckDB` zur genaueren Diagnose aufgerufen werden.

```
PROCEDURE Navigate (fromVersion  : VersionId;
                    direction    : NavigationDirection;
                    cursorVersion: VersionId;
                    VAR toVersion: VersionId;
                    VAR info     : ReturnInfo);
```

Mit Navigate kann auf dem Referenz- und Präzedenzgraphen einer Versionenmenge navigiert werden. fromVersion bezeichnet eine Ausgangsversion von der aus in einer mit direction spezifizierten Richtung entlang einer Kante gewandert werden soll. Für wiederholte Aufrufe von der gleichen fromVersion aus, ist zur Richtungssteuerung eine cursorVersion anzugeben. Die Identifikation der Version zu der gewandert wurde, wird in toVersion zurückgereicht.

fromVersion muss voll spezifiziert sein und bezeichnet eine Version, von der aus zu einer direkten Nachbarversion in einem der beiden Graphen zu navigieren ist.

Mit dem Parameter **direction** wählt der Benutzer den Graphen und die Navigationsrichtung.

Die Werte anyPredecessor, nextPredecessor, anyDescendant und nextDescendant dienen dem Navigieren auf dem Präzedenzgraphen. Mit anyPredecessor bzw. anyDescendant wird von fromVersion aus zu irgendeinem direkten Vorgänger bzw. Nachfolger navigiert. Um alle Vorgänger bzw. Nachfolger von fromVersion zu erhalten, wird Navigate wiederholt aufgerufen. Beim ersten Aufruf ist dazu direction = anyPredecessor bzw. = anyDescendant und bei jedem weiteren Aufruf direction = nextPredecessor bzw. = nextDescendant zu spezifizieren. Bei den next-Aufrufen ist im Parameter **cursorVersion** zusätzlich als Cursor die Identifikation der Version anzugeben, die beim vorhergehenden Aufruf erhalten wurde.

Werte anyReferencing, nextReferencing und referenced im Parameter direction erlauben ein Navigieren auf dem Referenzgraphen. Mit anyReferencing kann zu einer referenzierenden Version navigiert werden. Weitere Aufrufe von Navigate mit nextReferencing unter Angabe der zuletzt erhaltenen Version als Cursor (cursorVersion) liefern alle weiteren referenzierenden Versionen von fromVersion. Die von fromVersion referenzierte Version kann mit direction = referenced erhalten werden.

Im Parameter **toVersion** wird die Identifikation der Version, zu der navigiert wurde, zurückgegeben, sofern die Operation erfolgreich beendet wurde.

Hierüber informiert der Returnparameter **info** der mitteilt, ob navigiert wurde, ob keine toVersion gefunden wurde oder ob ein Fehler aufgetreten ist. Fehlermeldungen sind durch Aufrufe von CheckDB erhältlich.

```
PROCEDURE RetrieveRelational (granule      : Granule;
                              direction    : Direction;
                              order        : Order;
                              key          : NameType;
                              VAR tupleId  : TupleId;
                              VAR tupleData: Tuple;
                              VAR info     : ReturnInfo);
```

RetrieveRelational dient dem Abfragen von Information aus einer einzelnen Relation. Mit dem Parameter granule wird spezifiziert, mit welcher Genauigkeit die Relation, aus der gelesen wird, zu betrachten ist. Mit direction wird die Navigationsrichtung auf den Daten angegeben. order besagt, welches Ordnungskriterium für die Navigation massgebend ist. Falls in einer Schlüsselreihenfolge zu lesen ist, enthält Parameter key den Namen des massgebenden Schlüssels. In tupleId wird das Surrogat des gelesenen Tupels zurückgegeben und unter tupleData das Tupel selbst.

Im Parameter **granule** wird das Lesegranulat der Operation bestimmt. Der Benutzer kann wählen zwischen dem Lesen aller Versionenmengen der Relation (granule = set), allen Versionen einer Versionenmenge (granule = version) oder allen Tupeln einer Version, die zu einer bestimmten Relation gehören (granule = tuple).

Mit **direction** wird die Richtung bestimmt, in der gemäss der zugrundeliegenden Ordnung navigiert werden soll. Mögliche Werte sind first und last zur Absolutpositionierung, next und previous zur Relativpositionierung und current für Direktzugriffe.

Der Benutzer spezifiziert mit **order**, ob die Tupel einer systemdefinierten Reihenfolge unterliegen sollen (order = system), oder ob dafür ein Schlüssel massgebend ist (order = key).

Falls das Ordnungskriterium ein Schlüssel ist, muss dessen Name im Parameter **key** spezifiziert sein. Andernfalls ist der Wert von key belanglos.

Beim Aufruf der Prozedur muss **tupleId** in folgender Weise spezifiziert sein: tupleId.relNo muss immer die Relation bezeichnen, in der gelesen wird. Bei einer Absolutpositionierung mit first oder last müssen bei granule = set keine weiteren Teile von tupleId angegeben werden. Es wird ein Tupel der ersten oder letzten Versionenmenge gesucht. Bei granule = version muss zusätzlich mit tupleId.setNo die Versionenmenge identifiziert werden, in der ein erstes oder letztes Tupel zu suchen ist. Entsprechend muss bei granule = tuple tupleId.verNo die Version bezeichnen, aus der das erste oder letzte Tupel zu lesen ist. Bei Relativpositionierungen mit next und previous muss als Cursor jeweils die letzte Versionenmenge, Version oder das letzte, gelesene Tupel angegeben sein. So muss beispielsweise bei granule = version die Versionenmenge und die letzte Version im Parameter tupleId übergeben werden. Bei einem Direktzugriff mit direction = current ist das Surrogat des zu lesenden Tupels voll in tupleId zu übergeben. Im Parameter tupleId wird in jedem Fall das voll spezifizierte Surrogat des gelesenen Tupels zurückgegeben, falls die Operation erfolgreich war.

tupleData muss beim Aufruf der Prozedur die Adresse eines Speicherplatzes enthalten, in den das gelesene Tupel vom System übertragen werden soll. Die einzelnen Attribute des Tupels werden dort gemäss der Definition im Datenbankschema eingetragen.

Über den Parameter **info** wird mitgeteilt, ob ein Tupel gelesen wurde, ob kein Tupel gefunden wurde oder ob ein Fehler aufgetreten ist. Ist ein Fehler aufgetreten, können Fehlermeldungen über Aufrufe von CheckDB abgerufen werden.

```
PROCEDURE RetrieveObjectOriented (granule      : Granule;
                                  direction    : Direction;
                                  VAR version  : VersionId;
                                  VAR tupleId  : TupleId;
                                  VAR tupleData: Tuple;
                                  VAR info     : ReturnInfo);
```

RetrieveObjectOriented dient dem objektorientierten Abfragen der Datenbank. Mit dem Parameter granule wird spezifiziert, mit welcher Genauigkeit die Objektklasse, aus der gelesen wird, zu betrachten ist. Mit direction wird die Navigationsrichtung auf den Daten angegeben. version bezeichnet eine Version, in der gelesen werden soll. In tupleId wird das Surrogat des gelesenen Tupels zurückgegeben und unter tupleData das Tupel selbst.

Im Parameter **granule** wird das Lesegranulat der Operation bestimmt. Der Benutzer kann wählen zwischen dem Lesen aller Versionenmengen einer Objektklasse (granule = set), allen Versionen einer Versionenmenge (granule = version) oder allen Tupeln einer Version (granule = tuple).

Mit **direction** wird die Richtung bestimmt, in der navigiert werden soll. Mögliche Werte sind first und last zur Absolutpositionierung, next und previous zur Relativpositionierung, current für Direktzugriffe sowie subordinate und superordinate zum Navigieren entlang H-Referenzen.

version muss in folgender Weise spezifiziert übergeben werden: Bei einer Absolutpositionierung mit `first` oder `last` muss bei `granule = set` nur die Objektklasse angegeben sein (`version.objectClass`), bei `granule = version` zusätzlich die Versionenmenge (`version.objectSet`) und bei `granule = tuple` auch der dritte Teil zur Identifikation der Version (`version.version`). Bei Relativpositionierungen mit `next` und `previous` muss jeweils ein Teil mehr spezifiziert sein, also bei `granule = set` die Versionenmenge und bei `granule = version` die Versionenmenge und die Version.

Bei `granule = tuple`, beim Direktzugriff und beim Navigieren entlang H-Referenzen wird zusätzlich der voll spezifizierte Parameter **tupleId** benötigt. Das damit identifizierte Tupel wird beim Direktzugriff gelesen oder dient als Cursor für Relativpositionierungen innerhalb einer Version. Mit `direction = subordinate` bzw. `= superordinate` kann in einer H-Relation ein abhängiges bzw. das übergeordnete Tupel gelesen werden. In `tupleId` ist mindestens immer eine Relation zu spezifizieren (`tupleId.relNo`). Sie ist bei einer Absolutpositionierung die erste Relation der Objektklasse, aus der gelesen wird, oder dient bei allen anderen Zugriffsarten als Cursor. Bei erfolgreichem Ausführen der Operation wird in den Parametern `version` und `tupleId` in jedem Fall die voll spezifizierte Version und das Surrogat des gelesenen Tupels zurückgegeben.

tupleData muss beim Aufruf der Prozedur die Adresse eines Speicherplatzes enthalten, in den das gelesene Tupel vom System übertragen werden soll. Die einzelnen Attribute des Tupels werden dort gemäss der Definition im Datenbankschema eingetragen.

Der Parameter **info** teilt mit, ob ein Tupel gelesen wurde, ob dabei die Relation innerhalb der Objektklasse gewechselt wurde, ob kein Tupel gefunden wurde oder ob ein Fehler aufgetreten ist. Fehlermeldungen können über Aufrufe von `CheckDB` abgefragt werden.

Eidg. Techn. Hochschule Zürich
Informatikbibliothek
ETH-Zentrum
CH-8092 Zürich