# Exposé

zur Studienarbeit

**”Hierarchische Versionierung in relationalen Datenbanken“**

von Karsten Lohse

Mai 2007

Betreuer: Prof. Ulf Leser

Arbeitsgruppe Wissensmanagement in der Bioinformatik
Institut für Informatik
Humboldt-Universität zu Berlin
Unter den Linden 6
10099 Berlin

## Einleitung

Die hierarchische Versionierung von relationalen Datenbanken wird unter anderem im Bereich des DataWarehousing eingesetzt, um von einem definierten Stand durch Veränderung verschiedener Parameter verschiedene neue Datenbankzustände (Vorausberechnungen) zu erzeugen. Dies ist auf unterschiedlichen Wegen möglich:

- Datenbank-Dump erstellen und in abgetrennter Umgebung die aktuelle Datenbank neu aufbauen
- Die Veränderungen in einer Transaktion durchführen, die am Ende mit einem Rollback zurückgesetzt wird
- hierarchische Versionierung, d.h. es werden die Änderungen zum aktuellen Stand der Datenbank ebenfalls in der Datenbank verwaltet. So ist es möglich zu jeder vorherigen Version zurückzukehren.

## Motivation

In einer fast unzählbaren Zahl von Anwendungen werden heute relationale Datenbanken sowohl in vielen Geschäftsprozessen, wie auch in der Forschung zur Speicherung von Informationen eingesetzt. Hierbei entsteht schnell eine große Flut von Daten, die in Datenbank-Managementsystemen verwaltet werden müssen. Um solche Datenmenge verwalten zu können, werden sogenannte DataWarehouses eingesetzt. Dies sind große Datenbanken die als „Datenlager“ für das Unternehmen dienen. DataWarehouses kommen unter anderem in großen Einzelhandelsketten und bei Großhändlern zum Einsatz. Alle Vorgänge werden dabei in ein DataWarehouse eingepflegt. Dies geschieht durch Abgleich der Datenbankmanagement-Systeme mit dem DataWarehouse. Dieser Prozess kann sowohl in Echtzeit, sowie als regelmäßiger Bulk-Import realisiert sein. Die so gesammelten Daten dienen dann zur Auswertung von Geschäftsprozessen.

Allerdings sollen damit nicht nur zurückliegende Prozesse analysiert werden, sondern es sollen auch operative Entscheidungen der Zukunft durch die Auswertung der Daten beeinflusst werden. Hierfür ist eine Vorausberechnung mit vielen verschiedenen Parametern notwendig. Diese Vorausberechnung beruht auf den aktuellen Daten. Um verschiedene Szenarien durchzuspielen, kann man nun auch die zugrunde liegende Datenbasis (leicht) verändern. Somit kann man die Auswirkungen von Änderungen in Hinblick auf die Vorausbetrachtung beobachten. Allerdings kann man dies nicht in der aktuellen Datenbank machen, da man sich sonst seine realen Daten zerstören würde. Es ist daher notwendig, die Daten der Vorausberechnung strikt von den realen Daten zu trennen.

Einige mögliche Verfahren sind unter Einleitung benannt.

Eine elegante Variante hierfür ist die der hierarchischen Versionierung. Bei diesem Verfahren kann man die verschiedenen Versionen in einem Baum darstellen. Somit hat jede Version genau einen Vorgänger, kann aber mehrere Nachfolger haben. Dieses Verfahren ist auch dann zu favorisieren, wenn mehrere Nutzer mit den gleichen Ursprungsdaten Berechnungen anstellen sollen.

## Entwicklung einer hierarchischen Versionierung

Ziel dieser Studienarbeit ist es, die bereits von Stefan Rieche entwickelten linearen Versionierungsstrategien (1) in hierarchische zu überführen.

Die bei der hierarchischen Versionierung entstehenden Versionen sind in einem Baum darstellbar. Um auf die einzelnen Versionen wieder zugreifen zu können, muss jeder Knoten des Baumes „effizient ansprechbar“ sein und es muss möglich sein, die Liste seiner Vorgänger zu ermitteln.

Eine einfache Art der Indizierung der entstehenden Versionen kann mit einem Pre-Post-Order-Index erreicht werden. Hierbei wird bei einem depth-first-Ablaufen des Baumes jeder Knoten bei seinem ersten und letzten Kontakt mit einer Zahl beschriftet.

Hier ist ein Beispiel zu sehen:

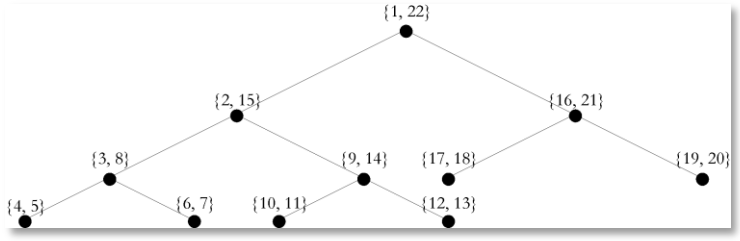



Abb. 1: Pre- Post- Order- Index 1

Eine solche Indizierung ist wichtig, um den entstehenden Baum in einer Datenstruktur repräsentieren zu können. Mit der Hilfe des Index ist es möglich, Vorgänger-Nachfolger-Beziehungen zu erkennen.

Um auf die Daten einer bestimmten Version zugreifen zu können, muss es eine Sicht geben, die durch Zugriff auf die Vorgängerversion und die gespeicherten Unterschiede bzw. Änderungen zur gewünschten Version, die Daten bereit stellen kann.

Wie schon bei Rieche (1) beschrieben ist, gibt es verschiedene grundlegende Arten, eine Versionierung zu realisieren.

- Versionierung mit Zeitstempel auf Tupelebene
  (Es wird für jedes Tupel ein Gültigkeitsintervall angegeben)
- Versionierung mit Deltas auf Tupelebene
  (Es wird gespeichert, welche Änderungen zwischen zwei Versionen bestehen)
- Versionierung auf Attributebene

Ziel dieser Studienarbeit ist es, die Möglichkeiten in Hinblick auf

- Redundanz der Speicherung von Informationen
- Sicherung von Fremdschlüsselbeziehungen
- Aufwand für den Zugriff (Lesen/Schreiben) auf die jeweils aktuellen Versionen
- Aufwand für den Zugriff (Lesen/Schreiben) auf alte Versionen
- Kompatibilität mit einem Mehrbenutzerbetrieb

zu untersuchen und zu bewerten.

Folgendes Beispiel soll die Versionierung verdeutlichen.

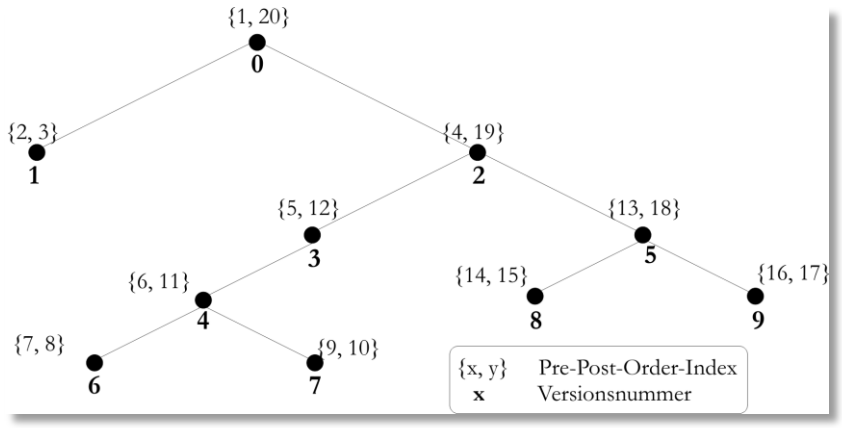
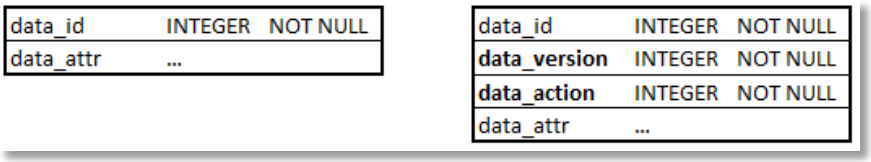



Abb. 2: Versionenbaum

Für die Verwaltung der Versionierungsinformationen werden in der Tabelle *data* zwei zusätzliche Attribute benötigt.



| data_id | INTEGER | NOT NULL |
|---|---|---|
| data_attr | ... | |

| data_id | INTEGER | NOT NULL |
|---|---|---|
| **data_version** | INTEGER | NOT NULL |
| **data_action** | INTEGER | NOT NULL |
| data_attr | ... | |

Abb. 3: Tabelle *data* (alt / neu)

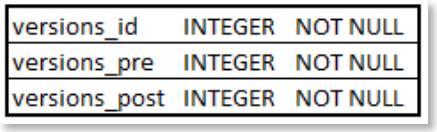
Im Attribut *data_version* wird die Versionsnummer verwaltet, in der das entsprechende Tupel geändert wird. Diese Änderung kann entweder das *Einfügen* oder das *Löschen* sein. Ein *Ändern* wird als *Löschen* und *Einfügen* simuliert. Diese Aktion wird im Attribut *data_action* als Integer gespeichert.

In der folgenden Abbildung ist ein Beispiel für den Inhalt einer *data* Tabelle zu sehen.

| data_id | data_version | data_action | data_attr |
|---|---|---|---|
| 1 | 0 | 1 | 1 |
| 2 | 0 | 1 | 2 |
| 3 | 0 | 1 | 3 |
| 2 | 1 | 2 | 2 |
| 2 | 1 | 1 | 20 |
| 1 | 2 | 2 | 1 |
| 4 | 2 | 1 | 4 |
| 5 | 3 | 1 | 5 |
| 6 | 4 | 1 | 6 |
| 2 | 4 | 2 | 2 |

Abb. 4: Beispiel Tabelle *data*

Die Werte für den Pre-Post-Order-Index werden in der neuen Tabelle *versions* gespeichert.



| versions_id | INTEGER | NOT NULL |
|---|---|---|
| versions_pre | INTEGER | NOT NULL |
| versions_post | INTEGER | NOT NULL |

Abb. 5: Tabelle *data*

Um beispielsweise die Version 4 wiederherstellen zu können, muss zuerst ermittelt werden, welche Versionen die Vorgänger der Version 4 sind. Hier im Beispiel sind dies 0, 2 und 3. Diese werden, zusammen mit der Nummer der

gewünschten Version (4) aufsteigend sortiert und damit die auszuführenden Aktionen aus der Tabelle *data* ausgelesen. Diese Aktionen werden nun ausgeführt. Die resultierende Tabelle ist die Tabelle der Version 4.

## vergleichende Messung

Abschließend werden die Zugriffsgeschwindigkeiten der neu entwickelten Versionierungsstrategien mit einem bereits bestehenden System (Oracle Workspace Manager) verglichen werden.
Hierbei sollen SELECTs, INSERTs, DELETEs und UPDATEs sowohl beim Zugriff auf die aktuelle Version, wie auch beim Zugriff auf unterschiedlich tief in der Hierarchie angesiedelten Versionen gemessen werden.

## Literaturverzeichnis

1. Rieche, Stephan. Diplomarbeit "Versionierung in relationalen Datenbanken". 2004.