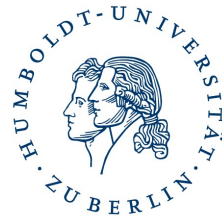


Studienarbeit

„Hierarchische Versionierung in relationalen Datenbanken“

Karsten Lohse



September 2007

Betreuer: Professor Dr. Ulf Leser

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

Erklärung

Hiermit erkläre ich, die vorliegende Studienarbeit selbstständig angefertigt zu haben. Es wurden nur die in der Arbeit ausdrücklich genannten Quellen und Hilfsmittel benutzt. Wörtlich oder sinngemäß übernommenes Gedankengut wurde als solches kenntlich gemacht.

Unterschrift:

Datum:

Inhaltsverzeichnis

1	Einleitung	9
1.1	Motivation	9
1.2	Ziel dieser Arbeit	10
1.3	Historie	10
1.4	Aufbau dieser Arbeit	10
2	Versionierungsstrategien	11
2.1	Unterschiede zwischen linearer und hierarchischer Versionierung	11
2.2	Arten der Versionierung	12
2.2.1	Versionierung mit Zeitstempel auf Tupelebene ohne Schattentabellen .	13
2.2.2	Versionierung mit Zeitstempel auf Tupelebene mit Schattentabellen .	13
2.2.3	Versionierung mit Deltas auf Tupelebene	13
2.2.4	Versionierung auf Attributebene	14
3	Entwicklung einer hierarchischen Versionierung	17
3.1	Einführung von Begriffen	17
3.2	Idee einer hierarchischen Versionierung	18
3.3	Verwaltung der Versionen	19
3.3.1	Tabelle <i>versions</i>	19
3.3.2	Erzeugung einer neuen Version	20
3.4	Veränderungen an den Datentabellen	20
3.5	Zugriff auf die versionierte Tabelle	21
3.6	Datenmanipulationen	22
3.6.1	<i>insert_data</i> -Trigger	22
3.6.2	<i>delete_data</i> -Trigger	23
3.6.3	<i>update_data</i> -Trigger	23
4	Beispiel einer Versionierung	25

4.1	Ausgangstabelle	25
4.1.1	Struktur	25
4.1.2	Inhalt	25
4.2	Umgewandelte und neu erzeugte Tabellen - Schritt 1	26
4.2.1	Struktur	26
4.2.2	Inhalt	26
4.2.3	Veränderungen	27
4.3	Umgewandelte und neu erzeugte Tabellen - Schritt 2	27
4.3.1	Inhalt	27
4.3.2	Veränderungen	29
4.4	Umgewandelte und neu erzeugte Tabellen - Schritt 3	29
4.4.1	Inhalt	29
4.5	Inhalte der angelegten Versionen der Tabelle <i>data</i>	30
4.5.1	Versionenbaum	30
4.5.2	Version 0	30
4.5.3	Version 1	30
4.5.4	Version 2	31
4.5.5	Version 3	31
4.5.6	Version 4	31
4.5.7	Version 5	31
5	Vergleichende Messung	33
5.1	Swissprot	33
5.2	Testumgebung	33
5.3	Einfügen der Daten	34
5.4	Zugriff auf Versionen	36
5.4.1	Anfrage ganzer Relationen	37
5.4.2	Anfrage von Bereichen aus Relationen	38
5.5	Optimierungen	45
5.5.1	Alternative Ermittlung der Vorgängerversionen 1	45
5.5.2	Alternative Ermittlung der Vorgängerversionen 2	45
5.5.3	Indizierung und Partitionierung	45
5.5.4	Ergebnisse der Optimierungen	46
6	Fazit	47
6.1	Constraints	47

6.2	Erweiterungsmöglichkeiten	47
A	API	48
B	Algorithmus der Messung	53
C	Struktur von SwissProt	55

Kapitel 1

Einleitung

In einer fast unzählbaren Zahl von Anwendungen werden heute Datenbanken zur Speicherung von Informationen eingesetzt. Dies sind unter anderem objektorientierte Datenbanken, Datendateien, XML-Dateien oder relationale Datenbanken.

1.1 Motivation

Relationale Datenbanken werden in allen Bereichen des täglichen Geschäftes eingesetzt. Als Beispiel möchte ich einmal den Lebensmittelhandel heraus greifen.

In einem modernen Supermarkt entstehen ständig große Mengen an Daten. Dies können unter anderem Bestelldaten, Wareneingangsdaten und Kassierdaten sein. Die entstehenden Daten werden heute meist in einem relationalem Datenbankmanagementsystem abgelegt.

In diesem Datenbankmanagementsystem können allerdings nicht alle Daten gespeichert werden, die je erfasst wurden. Dies würde schnell die Effizienz der operativen Systeme beeinträchtigen. Stattdessen werden die Daten in ein so genanntes DataWarehouse übertragen. Dies ist ein (meist zentrales) Datenlager, in dem alle entstandenen Daten archiviert werden. Dieser Prozess kann sowohl in Echtzeit sowie als regelmäßiger Bulk-Import realisiert sein. Die so gesammelten Daten dienen dann beispielsweise zur Analyse von Geschäftsprozessen.

Dabei sollen damit nur zurück liegende Prozesse analysiert werden, sondern es sollen auch operative Entscheidungen für die Zukunft unterstützt werden. Hierfür ist eine Vorausberechnung mit vielen verschiedenen Parametern notwendig. Diese Vorausberechnung beruht auf den aktuellen Daten. Um verschiedene Szenarien durchspielen zu können, will man aber eventuell auch die zugrunde liegende Datenbasis verändern.

Besser ist es, die Zukunftsbetrachtungen so auszuführen, dass sie die aktuellen und realen Daten nicht beeinflussen. Dies kann man mit verschiedenen Herangehensweisen realisieren. Eine oft praktizierte Lösung ist, einen *Dump* der Datenbank anzufertigen und diesen in einer abgetrennten Umgebung wieder in eine neue Datenbank zu überführen. In dieser kann dann die Zukunftsbetrachtung durchgeführt werden. Bei großen Datenbeständen, wie sie in DataWarehouses nicht selten sind, kann dies allerdings sehr zeitaufwändig werden.

Die Veränderungen an den Daten kann man auch in einer Transaktion ausführen. Dies wird schließlich mit einem *Rollback* beendet. Die eigentlichen Daten sind dadurch nicht verändert worden. Dies führt zum Einen dazu, dass einige Tabellen mit einem *Lock* versehen werden, zum Anderen aber auch dazu, dass die Aktionen der Zukunftsbetrachtung bei einem erneuten

Aufruf auch erneut durchgeführt werden müssen. Dies ist zeitaufwändig und fehleranfällig.

Eine elegantere Möglichkeit ist es, die Betrachtungen in einer auf den aktuellen Daten aufbauenden, aber logisch von ihr getrennten Version der Datenbank durchzuführen. Hierfür ist eine Versionierung der Datenbank notwendig.

Bei einer versionierten Datenbank wird nicht nur die aktuelle Version der Datenbank in einem DBMS verwaltet, sondern es werden durch geeignete Mechanismen unterschiedliche Versionen der Datenbank parallel zur Verfügung gestellt. Diese können sich – je nach Wunsch und Konfiguration – gegenseitig beeinflussen oder komplett autark sein.

Es wird zwischen linearer und hierarchischer Versionierung unterschieden. Genau genommen ist die lineare Versionierung ein Sonderfall der hierarchischen Versionierung. Die genauen Unterschiede werden im Abschnitt 2 „Unterschiede zwischen linearer und hierarchischer Versionierung“ im Kapitel 2 „Versionierungsstrategien“ (auf Seite 11) erläutert.

1.2 Ziel dieser Arbeit

Das Ziel dieser Studienarbeit ist die Entwicklung einer leistungsfähigen hierarchischen Versionierungsstrategie, die auch den Zugriff von Altanwendungen ermöglicht, die die Versionierung nicht unterstützen. Die zu entwickelnde Strategie soll mit den Daten von SwissProt [10] getestet werden. Die Performance soll durch Vergleichsmessung mit einer Datenbank ohne Versionierung und mit dem Oracle Workspace Manager verglichen werden.

1.3 Historie

Diese Arbeit baut auf der Diplomarbeit[1] vom Stephan Rieche mit dem Titel „Versionierung in relationalen Datenbanken“ aus dem Jahre 2004 auf. Die dort untersuchten Ansätze werden auf ihre Tauglichkeit für die hierarchische Versionierung hin untersucht und eine geeignete Strategie daraus entwickelt. Die Infrastruktur, die Stephan Rieche entwickelt hat, soll dabei genutzt und evtl. umgebaut werden. Dies betrifft vor allem das Tool *swissparse* zum Einfügen von Daten in die Datenbank.

1.4 Aufbau dieser Arbeit

Im Kapitel 2 werden mögliche Versionierungsstrategien vorgestellt. Das Kapitel 3 zeigt die Implementation und Funktionsweise der neu entwickelten Versionierungsstrategie. Ein Beispiel im Kapitel 4 soll die Funktionsweise nochmals genauer darstellen. Die abschließende vergleichende Messung im Kapitel 5 stellt die Zugriffszeiten der zu vergleichenden Systeme gegenüber.

Kapitel 2

Versionierungsstrategien

Bei der Versionierung wird grundsätzlich zwischen zwei Arten unterschieden.

- lineare Versionierung
- hierarchische Versionierung

2.1 Unterschiede zwischen linearer und hierarchischer Versionierung

Bei der linearen Versionierung hat jede Version höchstens einen Nachfolger. Somit ist für jede Version sofort ersichtlich, welche Version ihr Vorgänger und welche, wenn vorhanden, ihr Nachfolger ist. In Abbildung 2.1 ist ein möglicher Versionenbaum, genauer eine Versionenkette dargestellt.

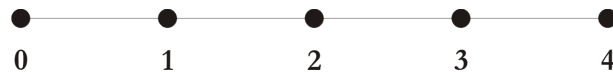


Abbildung 2.1: Lineare Versionierung

In Abbildung 2.2 ist ein möglicher Versionenbaum einer hierarchischen Versionierung dargestellt. Hier ist zu sehen, dass jede Version eine unterschiedliche Anzahl an Nachfolgern haben kann. Es handelt sich hier nicht zwingend um einen binären Baum. Das heißt ein Knoten kann auch mehr als 2 Kinder haben.

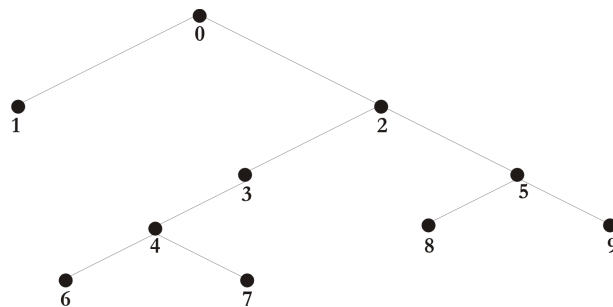


Abbildung 2.2: Hierarchische Versionierung

Bei der linearen Versionierung kann man eine automatische Erstellung der Versionen realisieren. Dies kann mit unterschiedlicher Frequenz als Reaktion auf bestimmte Ereignisse erfolgen. Die kann *transaktionskontinuierlich*, *anweisungskontinuierlich* oder *tupelkontinuierlich* erfolgen.

Transaktionskontinuierlich bedeutet, dass jeweils nach einer erfolgreich ausgeführten Transaktion eine neue Version angelegt wird.

Anweisungskontinuierlich bedeutet, dass nach der Abarbeitung jedes einzelnen SQL-Statements eine neue Version entsteht.

Bei der *tupelkontinuierlichen* Versionserstellung wird die Version durch Zeilentrigger ausgelöst, die bei jedem INSERT, UPDATE oder DELETE ausgeführt werden.

Die automatische Versionserstellung ist bei der hierarchischen Versionierung nicht mehr möglich, da das Datenbankmanagementsystem nicht weiß, wann eine neue Verzweigung angelegt werden soll. Daher muss die Versionserstellung manuell unter Angabe der Elternversion erfolgen.

Eine weitere Einschränkung betrifft die Möglichkeit, Daten zu ändern. Die Änderung ist nur in den Versionen möglich, die in den Blättern des Versionenbaumes zu finden sind. Änderungen an Versionen, die durch innere Knoten dargestellt werden, würden auch deren Nachfolgeversionen beeinflussen und sind somit nicht möglich. Dies soll im folgenden Beispiel verdeutlicht werden, das den Versionenbaum aus Abbildung 2.2 verwendet:

- in Version 2 wird ein neues Tupel eingefügt (*id: 111, data: 'a'*)
- in Version 0 existiert kein Tupel mit der *id 111*
- in Version 0 wird ein neues Tupel eingefügt (*id: 111, data: 'b'*)
- beim Zugriff auf die Version 2 und alle Versionen darunter gibt es nun einen Konflikt beim Tupel mit der *id 111*, dieses hat nun 2 verschiedene Attributwerte für *data* ('a' und 'b')

2.2 Arten der Versionierung

Nach [1] wird in folgende drei Arten von Versionierungsstrategien unterschieden:

- Versionierung mit Zeitstempel auf Tupelebene
 - Ohne Schattentabellen
 - Mit Schattentabellen
- Versionierung mit Deltas auf Tupelebene
- Versionierung auf Attributebene

Diese Arten der (linearen) Versionierung werden im Folgenden etwas genauer beschrieben. Für die hierarchische Versionierung sind sie allerdings nicht alle geeignet.

2.2.1 Versionierung mit Zeitstempel auf Tupelebene ohne Schattentabellen

Im Falle der Versionierung mit Zeitstempel auf Tupelebene ohne Schattentabellen wird zu jedem Tupel noch dessen Gültigkeitsbereich angegeben. Hierfür müssen zwei neue Attribute in die Relation eingefügt werden.

Aus der Relation A

<u>A.id</u>	<u>A.attr</u>
...	...

Tabelle 2.1: Tabelle A vor der Versionierung

wird nach der Umwandlung dies

<u>A.id</u>	<u>A.attr</u>	<u>A.Beginn</u>	<u>A.Ende</u>
...

Tabelle 2.2: Tabelle A nach dem Einfügen der Versionierungsattribute

$A.Beginn$ gibt an, in welcher Version das Tupel eingefügt wurde. $A.Ende$ hingegen gibt an, bis zu welcher Version es in der Tabelle enthalten war. Ist es noch in der letzten (aktuellen) Version gültig, hat $A.Ende$ den Wert -1 .

Somit können Anwendungen, die nichts von der Versionierung wissen, mit dem Zugriff auf eine View, die so definiert ist

```

1 CREATE VIEW A AS SELECT A_neu.id , A_neu.attr
2 FROM A_neu WHERE A_neu.Ende = -1;
```

auf die aktuellen Daten der Tabelle A zugreifen. Mit einer ähnlich gestalteten View ist es auch möglich, den Zugriff auf eine beliebige Version zu ermöglichen.

2.2.2 Versionierung mit Zeitstempel auf Tupelebene mit Schattentabellen

Bei der Versionierung mit Zeitstempel auf Tupelebene mit Schattentabellen wird nicht die Original-Relation verändert, sondern es wird eine zusätzliche Relation für jede Tabelle angelegt. Diese neue Tabelle hat den gleichen Aufbau wie die veränderte Tabelle aus der Versionierung mit Zeitstempel auf Tupelebene ohne Schattentabellen (Tabelle 2.2)

2.2.3 Versionierung mit Deltas auf Tupelebene

Bei der Versionierung mit Deltas auf Tupelebene werden nicht die kompletten Versionen, sondern nur ihre Änderungen gespeichert. Welche Änderungen gespeichert werden, wird im Folgenden beschrieben.

$KV_R(v_n)$ bezeichnet im Folgenden die komplette Version der Relation R mit der Versionsnummer n . Die Ursprungsversion ist v_0 . Die Unterschiede zwischen den Versionen i und j werden mit $\Delta_R(v_i, v_j)$ bezeichnet.

Im Allgemeinen kann man dabei vier Strategien unterscheiden:

VS-1

$$KV_R(v_n) = KV_R(v_0) + \Delta_R(v_0, v_1) + \dots + \Delta_R(v_{n-2}, v_{n-1}) + \Delta_R(v_{n-1}, v_n)$$

Diese Strategie ist vorwärtsorientiert. Das heißt, es wird die Ursprungsversion abgespeichert. Wird eine neue Version eingefügt, müssen die Unterschiede von dieser zu ihrer direkten Vorgängerversion gespeichert werden. Die n -te Version erhält man, indem man von der Ursprungsversion ausgehend alle Deltas aller Vorgängerversionen nacheinander anwendet.

VS-2

$$KV_R(v_n) = KV_R(v_0) + \Delta_R(v_0, v_n)$$

Auch diese Strategie ist vorwärtsorientiert. Bei ihr werden nicht die Deltas zur unmittelbaren Vorgängerversion, sondern die Deltas zur Ursprungsversion abgespeichert. Dadurch kann man jede Version über maximal zwei Versionen erzeugen. Die dabei entstehenden Deltas können schnell recht groß werden.

VS-3

$$KV_R(v_0) = KV_R(v_n) + \Delta_R(v_n, v_{n-1}) + \dots + \Delta_R(v_2, v_1) + \Delta_R(v_1, v_0)$$

Bei dieser Strategie wird die immer die letzte (aktuelle) Version gespeichert. Für jede Vorgängerversion wird immer das Delta zu deren Vorgängerversion gespeichert. Daher nennt man dieses Verfahren rückwärtsorientiert.

VS-4

$$KV_R(v_0) = KV_R(v_n) + \Delta_R(v_n, v_0)$$

Auch hierbei handelt es sich um ein rückwärtsorientiertes Verfahren. Es wird aber zu jeder Vorversion ein Delta zur letzten (aktuellen) Version gespeichert. Das heißt aber auch, dass bei jeder Änderungen an der aktuellen Version die Deltas aller Vorgängerversion verändert werden müssen.

2.2.4 Versionierung auf Attributebene

Da sich bei Änderungen in Tupeln nicht unbedingt immer alle Attributwerte ändern, ist es auch vorstellbar, nur die geänderten Attributwerte zu speichern.

Um die Veränderungen auf Attributebene durchführen zu können, muss die Originalrelation in Teilrelationen zerlegt werden – eine für jedes Attribut, das nicht zum Primärschlüssel gehört. Diese Relationen enthalten den Primärschlüssel, ein Attribut und die Attribute für Beginn und Ende der Gültigkeit des Tupels. Die folgenden Tabellen sollen dies verdeutlichen.

<u>A.id</u>	A.attr1	A.attr2
...

Tabelle 2.3: Tabelle A vor der Versionierung

<u>A_1.id</u>	A_1.attr1	A_1.beginn	A_1.ende
...

Tabelle 2.4: Teil-Tabelle A_1 nach der Zerlegung

<u>A_2.id</u>	A_2.attr2	A_2.Beginn	A_2.Ende
...

Tabelle 2.5: Teil-Tabelle A_2 nach der Zerlegung

Die Primärschlüssel $A_1.id$ und $A_2.id$ der neuen Relationen sind mit dem Primärschlüssel der alten Relation identisch.

Kapitel 3

Entwicklung einer hierarchischen Versionierung

Wie schon (weiter oben) beschrieben, gibt es verschiedene Strategien, wie man die Unterschiede der einzelnen Versionen im Datenbankmanagementsystem darstellt. Ich habe mich für die Speicherung der Deltas zwischen zwei Versionen entschieden. Hierbei werden in der Repräsentation der Kind-Version lediglich die Änderungen zur Vorgängerversion gespeichert. Dies hat den Vorteil, dass die Daten, die in einer zeitigen Version geändert oder eingefügt wurden, nicht in jeder nachfolgenden Version erneut gespeichert werden müssen. Allerdings muss man, um die Daten einer gewünschten Version wiederherstellen zu können, die Änderungen jede ihrer Vorgängerversionen ausführen.

3.1 Einführung von Begriffen

Unter einem *Versionenbaum* versteht man einen gerichteten Baum, in dem alle existierenden Versionen und ihre Beziehungen untereinander ersichtlich sind. Ein Beispiel findet sich in Abbildung 3.1 auf Seite 18. Gespeichert wird allerdings nicht der Baum selbst, sondern eine Repräsentation des Baumes in der Relation *versions*. Deren Struktur ist in Tabelle 3.1 auf Seite 19 zu sehen.

Unter *inneren Knoten* eines Baumes versteht man die Elemente (Knoten) eines Baumes, von denen Kinder existieren. Diese Knoten haben also mindestens einen Nachfolger. Im Beispiel in Abbildung 3.1 auf Seite 18 sind das die Konten *0*, *2*, *3*, *4* und *5*.

Unter *Blätter* eines Baumes versteht man die Elemente (Knoten) eines Baumes, von denen keine Kinder existieren. Das heißt diese Knoten haben keinen Nachfolger. Im Beispiel in Abbildung 3.1 auf Seite 18 sind das die Konten *1*, *6*, *7*, *8* und *9*.

Die *Vorgänger-Versionen* einer Version (eines Knotens) sind alle Versionen, die auf dem direkten Weg von der entsprechenden Version zur Wurzel des Baumes passiert werden. Im Beispiel in Abbildung 3.1 auf Seite 18 sind die Vorgänger-Versionen von Version *4* die Versionen *3*, *2* und *0*.

Nachfolger-Versionen von einer Version (einem Knoten) sind alle Versionen, die sich im Baum unterhalb der entsprechenden Version befinden. Im Beispiel in Abbildung 3.1 auf Seite 18 sind die Nachfolger-Versionen von Version *4* die Versionen *6* und *7*.

Unter der *Versions-Nummer* versteht man die eindeutige Repräsentation einer Version im Versionenbaum.

Ein *Pre-Post-Order-Index* ist eine einfache, aber effiziente Art der Indizierung eines Baumes. Hierbei wird bei einem depth-first-Ablauf des Baumes jeder Knoten bei seinem ersten und letzten Kontakt mit einer steigenden Zahl beschriftet.

Hier ist ein Beispiel zu sehen:

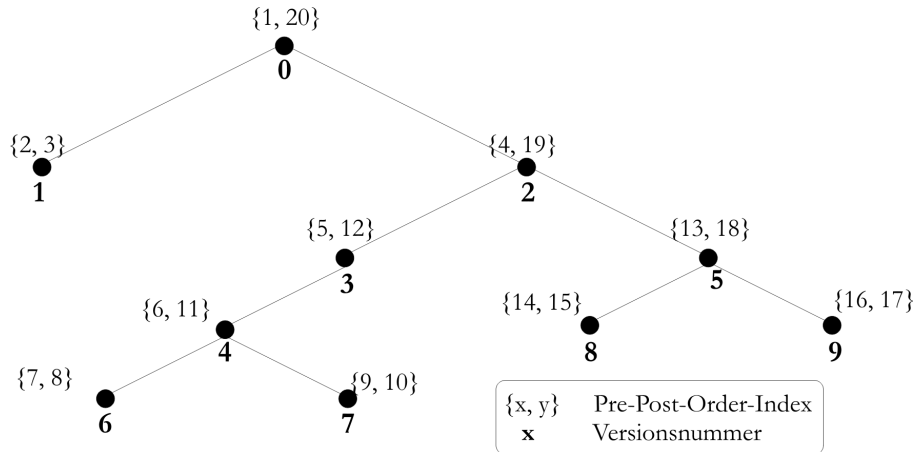


Abbildung 3.1: Pre-Post-Order-Index

Eine solche Indizierung ist notwendig, um effizient die Vorgänger-Nachfolger-Beziehungen der existierenden Versionen ermitteln zu können.

Beim Start einer neuen Session wird dieses Package initialisiert. Dabei wird die aktuelle Version auf die Version 0 gesetzt. Ist eine andere Start-Version gewünscht, so muss dies in der Zeile 191 des Package eingetragen werden.

Bei jedem Aufruf der *set_version* wird die Variable *aktuelle_version* gesetzt und die zu der entsprechenden Version gehörenden Werte für Pre- und Post-Order-Index in die Variablen *actual_pre* und *actual_post* geschrieben.

3.2 Idee einer hierarchischen Versionierung

Ziel einer hierarchischen Versionierung ist es, effizient auf alle existierenden Versionen zugreifen zu könne, ohne das spürbare Performaceverluste beim Zugriff auf tief im Versionenbaum liegende Versionen zu erkennen sind.

Ausgehend von einer nicht versionierten Tabelle sieht man, dass einige Bedingungen unbedingt zu beachten sind. Zum Beispiel muss es ausgeschlossen sein, dass es zu Inkonsistenzen in den Daten kommt. Da sämtliche Daten (aller existierenden Versionen) in einer Datentabelle gespeichert werden, ist es notwendig, eine Verwaltungsinformationen zu den Tupeln hinzuzufügen. Dies ist vor allem die Information, zu welcher Version das entsprechende Tupel gehört. Wenn in einer Version, die nicht die Wurzel des Versionenbaumes darstellt, ein Tupel gelöscht werden soll, so darf dieses nicht einfach aus der Tabelle entfernt werden. Dies würde das entsprechende Tupel auch in allen anderen Versionen mit löschen. Dies ist allerdings nicht gewünscht. Somit muss ein zu löschendes Tupel lediglich als gelöscht gekennzeichnet werden. Auch hier ist wieder eine zusätzliche Verwaltungsinformation notwendig. Es muss gespeichert

werden, welche Aktion mit einem Tupel durchgeführt wurde. Die kann das Einfügen eines neuen Tupels oder das Löschen eines bestehenden Tupels sein. Diese Information, verbunden mit der Versionsnummer, in der diese Änderung gültig ist, wird mit in das entsprechende Tupel geschrieben. Es ist vollkommen ausreichend, wenn man die Aktionen Einfügen und Löschen einführt. Ein Ändern der Daten kann über ein Löschen und anschließendes Einfügen abgebildet werden.

Um die Menge der gespeicherten Informationen so gering wie möglich zu halten, werden lediglich die Deltas zur Vorgängerversion gespeichert. Dies bedeutet aber auch, dass beim Zugriff auf eine Version etwas Rechenaufwand notwendig ist. Ausgehen von der gewünschten Version muss eine Versionenkette mit den jeweiligen Vorgängerversionen ermittelt werden. Dies kann mittels beispielsweise unter Verwendung des Pre-Post-Order-Index erfolgen.

Beginnend mit der Ursprungsversion (Wurzel des Versionenbaumes) wird nun, der Versionenkette entlang, jede Zwischenversion erzeugt, bis schließlich die gewünschte Version berechnet wird.

Diese wird, ohne den Verwaltungsinformationen, als View zur Verfügung gestellt. Somit ist es möglich, dass auch Altanwendungen, die keine Kenntnis von der Versionierung haben, auf diese Daten zugreifen können. Alle Anfragen werden an diese View gestellt und intern durch das Datenbankmanagementsystem umgewandelt.

Die genaue Funktion der hier entwickelten Versionierungsstrategie wird im Folgenden näher erläutert.

3.3 Verwaltung der Versionen

Bei der linearen Versionierung ist es nicht notwendig, die Vorgänger-Nachfolger-Beziehung zwischen den einzelnen Versionen abzuspeichern, da jede neue Version automatisch die nächste Versions-Nummer erhält. Somit ist klar, dass zum Beispiel die Version 4 der Nachfolger der Version 3 ist. Bei der hierarchischen Versionierung hingegen ist es notwendig, die Vorgänger-Nachfolger-Beziehungen der einzelnen Versionen zu speichern. Diese sind in einem Baum darstellbar. Um auf die einzelnen Versionen wieder zugreifen zu können, muss jeder Knoten des Baumes „effizient ansprechbar“ sein. Das heißt, es muss möglich sein, jede Version ohne große Zeitverzögerung auszurufen zu können. Um die gewünschte Version berechnen zu können, muss es möglich sein, die Liste der Vorgängerversionen ermitteln zu können. Dies ist notwendig, da zu jeder Version lediglich die Veränderungen zur Vorgänger-Version gespeichert werden. Auch muss man ermitteln können, ob es sich um ein Blatt oder um einen Knoten handelt. Mittels des Pre-Post-Order-Index ist dies möglich.

3.3.1 Tabelle *versions*

Die Versionen werden in der Tabelle *versions* verwaltet, die folgende Struktur besitzt.

versions_id	INTEGER	NOT NULL
versions_pre	INTEGER	NOT NULL
versions_post	INTEGER	NOT NULL

Tabelle 3.1: Tabelle *versions*

Jede erzeugte Version wird in diese Relation automatisch mit ihrer Versions-Nummer und den Werten des Pre-Post-Order-Index eingetragen.

Initialisiert wird diese Tabelle beispielsweise mit dem folgenden Eintrag: $(0, 0, 10)$. Hierbei wird der Version 0 der Pre-Index 0 und der Post-Index 10 zugeordnet.

3.3.2 Erzeugung einer neuen Version

Die PL/SQL-Prozedur *make_child(parent_version integer)* erzeugt einen neuen Knoten unterhalb der als Parameter übergebenen Version *parent_version*.

Es wird die größte bisher vergebene Versionsnummer ermittelt. Aus dieser wird die Versionsnummer für die neue Version ermittelt. Der Pre-Index und der Post-Index von *parent_version*, also der Eltern-Version, wird ermittelt. Es wird festgestellt, ob die Eltern-Version (*parent_version*) schon eine Nachfolgerversion besitzt. Ist dies der Fall, hat die neue Version bereits eine Nachbarversion. Wenn Pre- und Post-Index der neuen Version ermittelt sind, wird geprüft, ob die neue Version mit diesen Daten eingefügt werden kann. Ist dies nicht der Fall, muss zuerst der gesamte Index erweitert werden. Hierzu wird automatisch die Prozedur *make_tree_bigger(faktor integer)* mit dem Faktor 10 aufgerufen. Dabei wird in der Tabelle *versions* jeder Wert für Pre- und Post-Index mit dem Faktor 10 multipliziert. Somit werden wieder Zwischenräume zwischen den Indizes geschaffen. Ein weiteres Einfügen von Versionen ist damit möglich. Dies kann solange gemacht werden, bis beim größtem Index, also dem Post-Index der Ursprungsversion, der Wert von *MaxInt* erreicht wird.

Anschließend wird die Prozedur *make_child(parent_version integer)* erneut aufgerufen. Die Versions-Nummer der neu erzeugten Version wird ausgegeben.

Sie kann auch mit der Funktion *show_last_version* ermittelt werden.

3.4 Veränderungen an den Datentabellen

Für die Betrachtung der entwickelten hierarchischen Versionierung nehmen wir folgende Ausgangstabelle *data* an:

data_id	INTEGER	NOT NULL
data_attr	...	

Tabelle 3.2: Tabelle *data* (alt)

Hierbei handelt es sich bei der Spalte *data_id* um den Primärschlüssel der Tabelle. Der Primärschlüssel der Tabelle kann auch mehrteilig sein. Dies ist dann allerdings in der Triggern und in der Generatorfunktion auch zu berücksichtigen. Das Attribut *data_attr* steht hier nur exemplarisch für eine unbekannte Anzahl an Spalten von beliebigem Typ.

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

data_id	INTEGER	NOT NULL
data_attr	...	
data_version	INTEGER	NOT NULL
data_action	INTEGER	NOT NULL
data_counter	INTEGER	NOT NULL

Tabelle 3.3: Tabelle *data* (neu)

Im Attribut *data_version* wird die Versionsnummer abgelegt, zu der das entsprechende Tupel gehört. Im Feld *data_action* wird die mit dem Tupel durchgeführte Aktion als INTEGER-Wert gespeichert. Ein *Einfügen* wird mit der Zahl 1, ein *Löschen* mit der Zahl 2 dargestellt. Ein *Ändern* wird als *Löschen* und *Einfügen* simuliert. In der Spalte *data_counter* wird jeder Aktion eine Sequenznummer als Zeitstempel zugeordnet. Dies wird benötigt, um später zu prüfen, ob es spätere Aktionen für ein bestimmtes Tupel in einer Version gibt. Soll eine bereits bestehende Tabelle für die Versionierung ergänzt werden, so müssen diese drei Spalten eingefügt werden und mit folgenden Standardwerten gefüllt werden:

Attribut	Wert	Bemerkung
data_version	0	Ursprungsversion
data_action	1	Aktion <i>Einfügen</i>
data_counter	0	hier ist es wichtig, dass die nächste vergebene Zahl größer als die Initialisierung ist

Tabelle 3.4: Initialwerte

Zusätzlich sollte auch noch die Tabelle umbenannt werden.

3.5 Zugriff auf die versionierte Tabelle

Beim Connect mit der Datenbankinstanz wird für jede Session eine neue Instanz des Packages *h_versions* erzeugt. Bei dessen Initialisierung wird die Variable *actual_version* auf 0 gesetzt, sowie die Variablen *actual_pre* und *actual_post* mit den Werten für die Version 0 gesetzt.

Um die Version zu wechseln, muss folgender Befehl ausgeführt werden:

```
1   exec h_versions.set_version(3);
```

Dabei wird durch die Prozedur *set_version(version integer)* die Variable *actual_version* geändert und die Werte für *actual_pre* und *actual_post* aktualisiert.

Für jede versionierte Tabelle muss eine Generatorfunktion im Package *h_versions* enthalten sein, die die Daten der gewünschten Version aus den Informationen der Tabellen *hv_data* und *versions* generiert. In unserem Beispiel ist dies die Funktion *data_alt*.

Diese Funktion liefert als Rückgabewert eine Pipeline des Typs *data_tabellentyp*.

Die Funktion *data_alt* berechnet mittels Cursor alle Einträge der gewünschten Version und ihrer Vorgängerversionen. Der entsprechende SQL-Code ist im Listing des Package *h_versions* in den Zeilen 92 bis 135 zu sehen.

Die Berechnung muss in zwei Schritten erfolgen. Im ersten Schritt werden alle Einträge ermittelt, die in der gewünschten Version und deren Vorgängerversionen eingefügt werden (*data_action = 1*), ohne dass sie wieder gelöscht werden (*NOT EXISTS ... data_action = 2*).

Der zweite Schritt ermittelt für jedes Tupel, das in der Versionsgeschichte gelöscht und wieder eingefügt wird, mithilfe des Attributes *data_counter* den aktuellen Wert. Da Änderungen immer nur in Blättern möglich sind, ist somit garantiert, dass dies auch immer der gewünschte Wert ist. Diese Selektion wird durch Negation der Bedingung (*NOT EXISTS ... data_action = 2*) aus dem ersten Teil in (*EXISTS ... data_action = 2*) und Kombination mit (*NOT EXISTS ... data_3.data_counter > data_1.data_counter*) erreicht.

Die benötigten Daten (*data_id* und *data_attr*) des so erhaltenen Cursor werden in der Funktion *data_alt* solange in die Pipeline gesendet, wie noch Elemente im Cursor verfügbar sind. Somit sind die Daten der gewünschten Version der versionierten Tabelle *data* als Ergebnis der PL/SQL-Funktion erhältlich. Um auch den Zugriff von Altanwendungen zu ermöglichen, wird das Ergebnis noch in eine View namens *data* gepackt. Diese ist so definiert:

```
1      SELECT * FROM TABLE(h_versions.data_alt)
```

Somit können auch Anwendungen, die keinerlei Kenntnis von der Versionierung haben, noch auf die Daten aus einer beliebigen Version zugreifen. Es muss lediglich sichergestellt werden, dass die entsprechende Version zuvor ausgewählt wurde.

3.6 Datenmanipulationen

Um auch Altanwendungen die Manipulation von Daten zu ermöglichen, wurden Trigger implementiert, die die Manipulation von Daten in der View *data* abfangen und regelkonform in die versionsverwaltende Tabelle *hv_data* übertragen. Hierbei sind Trigger für INSERT, UPDATE und DELETE notwendig.

In den folgenden Triggern wird bei jedem Insert auch der Wert für *data_counter* mit geschrieben. Dieser wird durch die Funktion *next_seqnumber* erzeugt.

Die Trigger sind wie folgt definiert:

3.6.1 *insert_data*-Trigger

```
1      TRIGGER "INSERT_DATA" INSTEAD OF INSERT ON DATA
2      REFERENCING NEW AS added
3      BEGIN
4          IF (h_versions.has_child = 0)
5              THEN
6                  INSERT INTO hv_data (data_id, data_version,
7                                      data_action, data_attr,
8                                      data_counter)
9                  VALUES (:added.data_id, h_versions.actual_version,
10                         1, :added.data_attr, h_versions.next_seqnumber);
11             ELSE
12                 raise_application_error
13                 (-20001, 'Es_koennen_nur_Versionen_in_Blaettern_' ||
14                     'veraendert_werden!');
```

```

15     END IF ;
16 END;

```

Der *insert_data*-Trigger ersetzt das *Einfügen* in die View *data*. Nach der Prüfung, ob es sich um eine Blattversion handelt, wird in die Tabelle *hv_data* ein neuer Datensatz eingefügt, der die aktuelle Versionsnummer (*data_version*), die Original-Tupel-ID (*data_id*), die Aktionskennung 1 (für *Einfügen*) (*data_action*) und alle Tupel-Attribute (*data_attr*) enthält. Sollte es sich bei der aktuellen Version nicht um ein Blatt handeln, wird lediglich eine Fehlermeldung ausgegeben.

3.6.2 *delete_data*-Trigger

```

1  TRIGGER "DELETE_DATA" INSTEAD OF DELETE ON DATA
2  REFERENCING OLD AS deleted
3  BEGIN
4      IF (h_versions.has_child = 0)
5          THEN
6              DELETE FROM hv_data
7              WHERE data_id = :deleted.data_id
8              AND data_version = h_versions.actual_version
9              AND data_action = 2;
10
11             INSERT INTO hv_data
12                 (data_id, data_version, data_action, data_counter)
13             VALUES (:deleted.data_id, h_versions.actual_version, 2,
14                 h_versions.next_seqnumber);
15         ELSE
16             raise_application_error
17             (-20001, 'Es_koennen_nur_Versionen_in_Blaettern_||
18                 'veraendert_werden!');
19         END IF;
20 END;

```

Der *delete_data*-Trigger ersetzt das *Löschen* in der View *data*. Nach der Prüfung, ob es sich um eine Blattversion handelt, werden in der Tabelle *hv_data* zunächst alle Tupel gelöscht, die sich bereits auf die entsprechende Version, Primärschlüssel (*data_id*) und Aktion (*data_action* = 2) beziehen. Somit wird sichergestellt, dass keine doppelten Löscheinträge Probleme bei der Wiederherstellung der Daten machen. Danach wird ein neuer Datensatz eingefügt, der die aktuelle Versionsnummer (*data_version*), die Original-Tupel-ID (*data_id*) und die Aktionskennung 2 (für *Löschen*) (*data_action*) enthält. Eine Fehlermeldung wird ausgegeben, wenn sich das Löschen auf eine Version bezieht, die keine Blatt-Version ist.

3.6.3 *update_data*-Trigger

Der *update_data*-Trigger ist eine Kombination aus dem *delete_data*-Trigger und dem *insert_data*-Trigger.

Kapitel 4

Beispiel einer Versionierung

4.1 Ausgangstabelle

4.1.1 Struktur

data_id	INTEGER	NOT NULL
data_attr	VARCHAR2(200)	

Tabelle 4.1: Struktur der Tabelle *data*

Die Tabelle *data* enthält zwei Spalten. *data_id* ist der Primärschlüssel der Tabelle.

4.1.2 Inhalt

data_id	data_attr
1	'1.0'
2	'2.0'
3	'3.0'

Tabelle 4.2: Inhalt der Tabelle *data*

Diese Daten befinden sich bereits vor der Versionierung in der Tabelle *data*.

4.2 Umgewandelte und neu erzeugte Tabellen - Schritt 1

4.2.1 Struktur

<code>data_id</code>	INTEGER	NOT NULL
<code>data_attr</code>	VARCHAR2(200)	
<code>data_version</code>	INTEGER	NOT NULL
<code>data_action</code>	INTEGER	NOT NULL
<code>data_counter</code>	INTEGER	NOT NULL

Tabelle 4.3: Struktur der Tabelle *hv.data*

Es wurden die Versionsverwaltungsspalten *data_version*, *data_action* und *data_counter* eingefügt und die Tabelle wurde nach *hv.data* umbenannt.

<code>versions_id</code>	INTEGER	NOT NULL
<code>versions_pre</code>	INTEGER	NOT NULL
<code>versions_post</code>	INTEGER	NOT NULL

Tabelle 4.4: Struktur der Tabelle *versions*

Die Tabelle *versions* wurde neu erstellt.

4.2.2 Inhalt

<code>data_id</code>	<code>data_attr</code>	<code>data_version</code>	<code>data_action</code>	<code>data_counter</code>
1	'1.0'	0	1	0
2	'2.0'	0	1	0
3	'3.0'	0	1	0

Tabelle 4.5: Inhalt der Tabelle *hv.data*

Die vorhandenen Tupel der Tabelle *hv.data* wurden durch die initialen Versionsverwaltungsinformationen (*data_version* = 0, *data_action* = 1, *data_counter*=0) ergänzt.

<code>versions_id</code>	<code>versions_pre</code>	<code>versions_post</code>
0	0	10

Tabelle 4.6: Inhalt der Tabelle *versions*

In die Tabelle *versions* wurde die Ursprungsversion mit den Werten *versions_pre* = 0 und *versions_post* = 10 eingefügt.

Es werden nacheinander drei Unterversionen (1, 2, 3) von der Version 0 gebildet. Hierfür werden folgende Befehle ausgeführt:

```
exec h_versions.set_version(0);
```

```
exec h_versions.make_Child(0);
exec h_versions.make_Child(0);
exec h_versions.make_Child(0);
```

4.2.3 Veränderungen

Die aktuelle Version wird auf 2 gesetzt.

```
exec h_versions.set_version(2);
```

Es werden folgende Tupel eingefügt:

data_id	data_attr
4	'4.2'
5	'5.2'
6	'6.2'

Tabelle 4.7: neue Tupel für der Tabelle *data*

Die Inserts werden direkt an die View *data* gerichtet:

```
INSERT INTO data (data_id, data_attr) VALUES (4, '4.2'); (usw.)
```

Durch den Trigger *insert_data* wird dies abgefangen und stattdessen an die Tabelle *hv_data* weitergereicht. Dort wird folgendes ausgeführt:

```
INSERT INTO hv_data (data_id, data_version, data_action, data_attr)
VALUES (4, 2, 1, '4.2'); (usw.)
```

Dieses Statement wird wiederum durch den Trigger *insert_hv_data* erweitert zu:

```
INSERT INTO hv_data (data_id, data_version, data_action,
                    data_counter, data_attr)
VALUES (4, 2, 1, 1, '4.2'); (usw.)
```

4.3 Umgewandelte und neu erzeugte Tabellen - Schritt 2

4.3.1 Inhalt

Nach dem Einfügen sind folgende Tabellen im Datenbankmanagementsystem gespeichert.

data_id	data_attr	data_version	data_action	data_counter
1	'1.0'	0	1	0
2	'2.0'	0	1	0
3	'3.0'	0	1	0
4	'4.2'	2	1	1
5	'5.2'	2	1	2
6	'6.2'	2	1	3

Tabelle 4.8: Inhalt der Tabelle *hv.data*

versions_id	versions_pre	versions_post
0	0	100
1	30	60
2	70	80
3	86	93

Tabelle 4.9: Inhalt der Tabelle *versions*

Beim Anlegen der drei Unterversionen der Ursprungsversion 0 passierte folgendes:

Version	p_pre	p_post	n_post	Diff.	v_pre	v_post
0					0	10
1	0	10	0	10	$\lfloor 0 + 10 \cdot \frac{1}{3} \rfloor = 3$	$\lfloor 0 + 10 \cdot \frac{2}{3} \rfloor = 6$
2	0	10	6	4	$\lfloor 6 + 4 \cdot \frac{1}{3} \rfloor = 7$	$\lfloor 6 + 4 \cdot \frac{2}{3} \rfloor = 8$
3	0	10	8	2	$\lfloor 8 + 2 \cdot \frac{1}{3} \rfloor = 8$	$\lfloor 8 + 2 \cdot \frac{2}{3} \rfloor = 9$
<p>Eintrag 3 kann nicht geschrieben werden, <i>(neighbour_post = versions_post)</i> der Baum muss zuerst erweitert werden, die Werte <i>versions_pre</i> und <i>versions_post</i> der bestehenden Einträge werden mit dem Faktor 10 multipliziert</p> <p>Es ergeben sich folgende Einträge:</p>						
0					0	100
1					30	60
2					70	80
neuer Versuch des Einfügens der Version 3						
3	0	100	80	20	$\lfloor 80 + 20 \cdot \frac{1}{3} \rfloor = 86$	$\lfloor 80 + 20 \cdot \frac{2}{3} \rfloor = 93$

Tabelle 4.10: Anlegen der Versionen 1 bis 3 unter der Version 0

Spaltenname	Beschreibung
Version	Versions-Id
p_pre	Pre-Index der Eltern-Version
p_post	Post-Index der Eltern-Version
n_post	Post-Index der Geschwister-Version oder 0
Diff.	Differenz zwischen p_pre bzw. n_post und p_post
v_pre	Pre-Index der neuen Version
v_post	Post-Index der neuen Version

Tabelle 4.11: Erläuterung der Spaltennamen von Tabelle 4.10

4.3.2 Veränderungen

Es werden nacheinander 2 Unterversionen (4, 5) von der Version 2 gebildet.

Die aktuelle Version wird auf 4 gesetzt.

Das Tupel mit *data_id* = 5 wird gelöscht.

data_attr wird beim Tupel mit *data_id* = 6 auf *data_attr*='6.4' geändert.

Es werden folgende Tupel eingefügt:

data_id	data_attr
7	'7.4'

Tabelle 4.12: neue Tupel für der Tabelle *data*

4.4 Umgewandelte und neu erzeugte Tabellen - Schritt 3

4.4.1 Inhalt

data_id	data_attr	data_version	data_action	data_counter
1	'1.0'	0	1	0
2	'2.0'	0	1	0
3	'3.0'	0	1	0
4	'4.2'	2	1	1
5	'5.2'	2	1	2
6	'6.2'	2	1	3
5		4	2	4
5		4	2	5
6	'6.4'	4	1	6
7	'7.4'	4	1	7

Tabelle 4.13: Inhalt der Tabelle *hv.data*

versions_id	versions_pre	versions_post
0	0	100
1	30	60
2	70	80
3	86	93
4	73	76
5	77	78

Tabelle 4.14: Inhalt der Tabelle *versions*

4.5 Inhalte der angelegten Versionen der Tabelle *data*

4.5.1 Versionenbaum

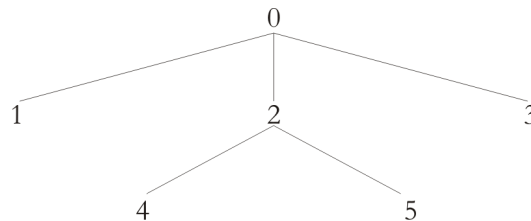


Abbildung 4.1: Versionenbaum

4.5.2 Version 0

data_id	data_attr
1	'1.0'
2	'2.0'
3	'3.0'

Tabelle 4.15: Inhalt der Tabelle *data* – Version 0

4.5.3 Version 1

data_id	data_attr
1	'1.0'
2	'2.0'
3	'3.0'

Tabelle 4.16: Inhalt der Tabelle *data* – Version 1

4.5.4 Version 2

data_id	data_attr
1	'1.0'
2	'2.0'
3	'3.0'
4	'4.2'
5	'5.2'
6	'6.2'

Tabelle 4.17: Inhalt der Tabelle *data* – Version 2

4.5.5 Version 3

data_id	data_attr
1	'1.0'
2	'2.0'
3	'3.0'

Tabelle 4.18: Inhalt der Tabelle *data* – Version 3

4.5.6 Version 4

data_id	data_attr
1	'1.0'
2	'2.0'
3	'3.0'
4	'4.2'
6	'6.4'
7	'7.4'

Tabelle 4.19: Inhalt der Tabelle *data* – Version 4

4.5.7 Version 5

data_id	data_attr
1	'1.0'
2	'2.0'
3	'3.0'
4	'4.2'
5	'5.2'
6	'6.2'

Tabelle 4.20: Inhalt der Tabelle *data* – Version 5

Kapitel 5

Vergleichende Messung

Neben der Entwicklung einer Methode für die hierarchische Versionierung war auch deren vergleichende Messung ein Ziel dieser Studienarbeit.

Zum Vergleich stehen eine Datenbank ohne Versionierung, die neu entwickelte Versionierungsstrategie dieser Studienarbeit und der Oracle Workspace Manager.

5.1 Swissprot

Da für einen aussagekräftigen Vergleich eine große Menge an Daten verarbeitet werden muss, wurde auf die Daten vom Swissprot [10] zurückgegriffen, die auch schon in [1] Verwendung fanden. Für das Einfügen wurden die Releases *37* und *44* verwendet. Das *Release 37* enthält 77977 und das *Release 44* 153871 Entries. Es wurden die Geschwindigkeiten für das Einfügen und das Zugreifen auf die beiden Versionen gemessen. Beim Einfügen von *Release 44* wurde mit dem Einfügetools stets geprüft, ob das einzufügende Tupel bereits in der Vorgänger-Version (*Release 37*) vorhanden war. Wenn es identisch vorhanden ist, wurde es nicht erneut eingefügt. War es mit Unterschiedlichen Daten enthalten, wurde ein Update ausgeführt.

Nach dem Einfügen von *Release 44* wurde eine neue Version unterhalb von *Release 44* erstellt. In dieser wurden 10000 Entries verändert. Dieses Release wird als *Release 44-1* bezeichnet. Unterhalb dieser Version wurde ebenfalls eine neue Version angelegt. Hier wurden 5000 Entries gelöscht. Dieses Release wird als *Release 44-2* bezeichnet.

5.2 Testumgebung

Alle Messungen wurden unter den gleichen Software- und Hardwarebedingungen durchgeführt. Allerdings kann es durch automatische Hintergrundprozesse des Betriebssystems zu leichten Schwankungen kommen.

Es wurde die Oracle Datenbank in Version *10g Release 2* (10.2) für Microsoft Windows (32-Bit) verwendet.

Gemessen wurde auf einen PC mit einem Intel Core2Duo 6600 mit 2,40 GHz und 2048 MB RAM. Als Betriebssystem wurde Windows Vista Ultimate eingesetzt.

Als Client wurde der *Oracle SQL Developer* eingesetzt.

5.3 Einfügen der Daten

Zur Messung der Einfügeschwindigkeit kam das von Stephan Rieche in [1] entwickelte Java-Programm *swissparse* zum Einsatz, das um die Funktionalität des Zugriffs auf die hierarchischen Versionen erweitert wurde. Das Programm liest die Datendateien der einzufügenden Relation und erstellt daraus die nötigen SQL-Statements zum Einfügen der Daten in die Datenbank. Dabei wird die Zeit gemessen, die für das Einfügen benötigt wird.

Die entsprechenden Geschwindigkeiten in eingetragenen Blöcken pro Sekunde, bzw. die Dauer des Eintragens sind in den Tabellen 5.1 und 5.2 sowie in den Diagrammen in den Abbildungen 5.1 und 5.2 zu sehen.

Etwaige Schwankungen in den Geschwindigkeiten sind u.a. durch Hintergrundaktivitäten des Betriebssystems zu begründen.

	Release 37	Release 44
ohne Versionierung	56,4	52,5
mit Versionierung	13,2	7,3
mit Workspace Manager	9,3	7,1

Tabelle 5.1: Entries pro Sekunde beim Einfügen

	Release 37	Release 44
ohne Versionierung	00:23:23	00:49:11
mit Versionierung	01:39:41	06:58:23
mit Workspace Manager	02:20:37	06:20:22

Tabelle 5.2: Dauer des Einfügens Einfügen in hh:mm:ss

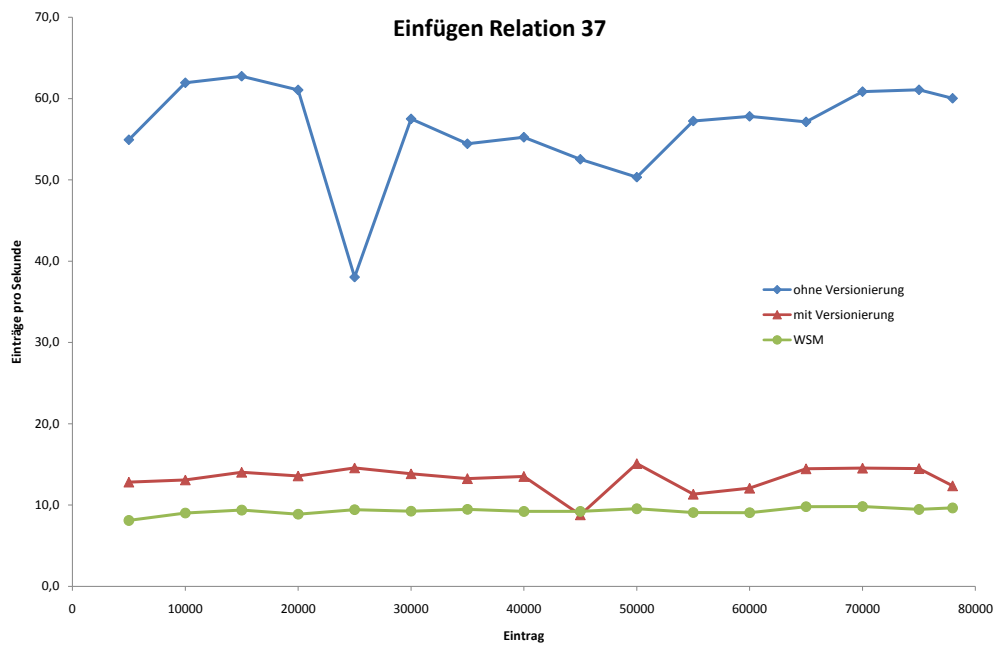


Abbildung 5.1: Geschwindigkeit des Einfügens des Release 37

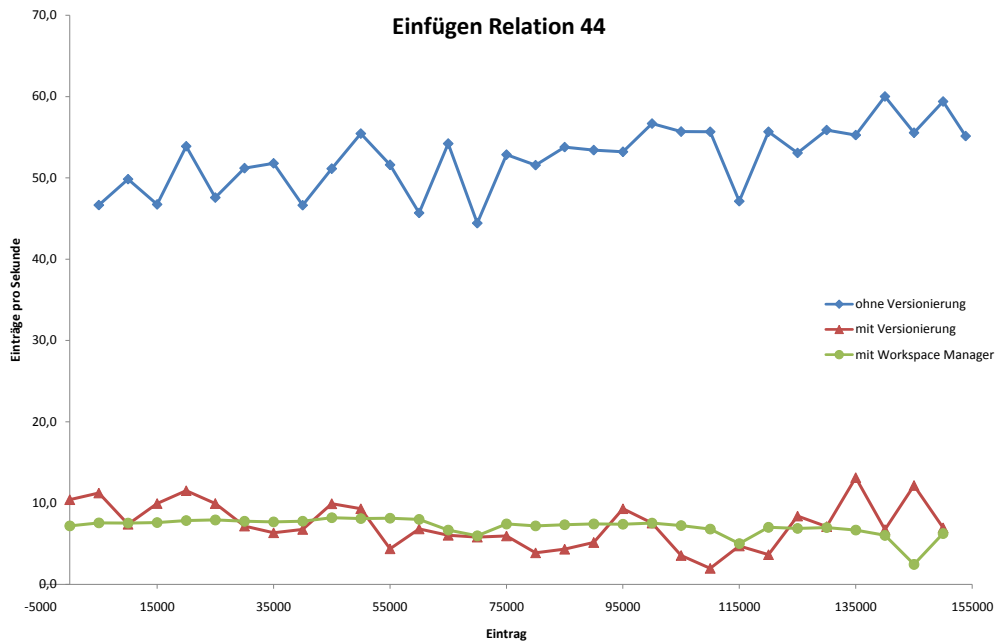


Abbildung 5.2: Geschwindigkeit des Einfügens des Release 44

Es ist hier zu sehen, dass das Einfügen in die Datenbank ohne Versionierung deutlich schneller vonstatten geht. Es handelt sich hier ungefähr um den Faktor 7. Das war so zu erwarten, da beim reinen Einfügen keine zusätzlichen Trigger ausgeführt werden müssen. Man sieht auch, dass die Leistungsfähigkeit des neu entwickelten Systems an die des Oracle Workspace Managers herankommt. Für das Einfügen des *Release 44* wird mehr Zeit benötigt, da dieses auch mehr einzutragende Entries enthält.

5.4 Zugriff auf Versionen

Beim Zugriff auf die Relationen einer Datenbank kann es durch den Buffer zu unterschiedlichen Zugriffszeiten für eine identische Anfrage kommen. Um diesen Effekt nahezu auszuschalten, wurde bei der Messung der Geschwindigkeiten bei lesenden Zugriffen 4 Einzelanfragen nacheinander gestellt und die Messergebnisse gemittelt. Da nur die reine Zeit für den Datenzugriff ermittelt werden sollte, wurde in Anlehnung an [1] wiederum mit Cursors gearbeitet. Somit konnte die Übertragungszeit der Ergebnisse zum Client eliminiert werden.

Es werden die folgenden SQL-Statements verwendet:

```

1      SQL 1: SELECT * FROM identification ;
2
3      SQL 2: SELECT * FROM identification a join
4              reference b on (a.id=b.id);
5

```

```

6      SQL 3: SELECT * FROM identification a join
7              reference b on (a.id=b.id) join
8              dt_table c on (a.id = c.id);
9
10     SQL 4: SELECT * FROM identification a join
11             reference b on (a.id=b.id) join
12             dt_table c on (a.id = c.id) join
13             sequence_line d on (a.id = d.id);
14
15     SQL 5: SELECT * FROM identification a
16             where a.sequence_length < 2;
17
18     SQL 6: SELECT * FROM identification a join
19             de b on (a.id=b.id)
20             where a.sequence_length < 2;
21
22     SQL 7: SELECT * FROM identification a join
23             de b on (a.id=b.id) join
24             os c on (a.id=c.id)
25             where a.sequence_length < 2;
26
27     SQL 8: SELECT * FROM identification a join
28             de b on (a.id=b.id) join
29             os c on (a.id=c.id) join
30             accession_numbers d on (a.id=d.id)
31             where a.sequence_length < 2;

```

5.4.1 Anfrage ganzer Relationen

Bei den Anfragen SQL 1 bis SQL 4 wurden die in Tabelle 5.3 und Abbildung 5.3 dargestellten Zeiten gemessen. Die Werte für die Messungen *ohne Versionierung* sind erwartungsgemäß kleiner als die für die Messungen mit den beiden Versionierungssystemen. Es ist auch hier zu sehen, dass die entwickelte Versionierung in etwa gleich schnell wie der Oracle Workspace Manager ist.

	ohne Versionierung	mit Versionierung	mit Workspace Manager
SQL 1	2957 ms	7617 ms	6041 ms
SQL 2	10227 ms	48156 ms	50996 ms
SQL 3	51656 ms	90070 ms	83467 ms
SQL 4	141770 ms	197426 ms	192471 ms

Tabelle 5.3: Dauer der Anfragen SQL 1 bis SQL 4

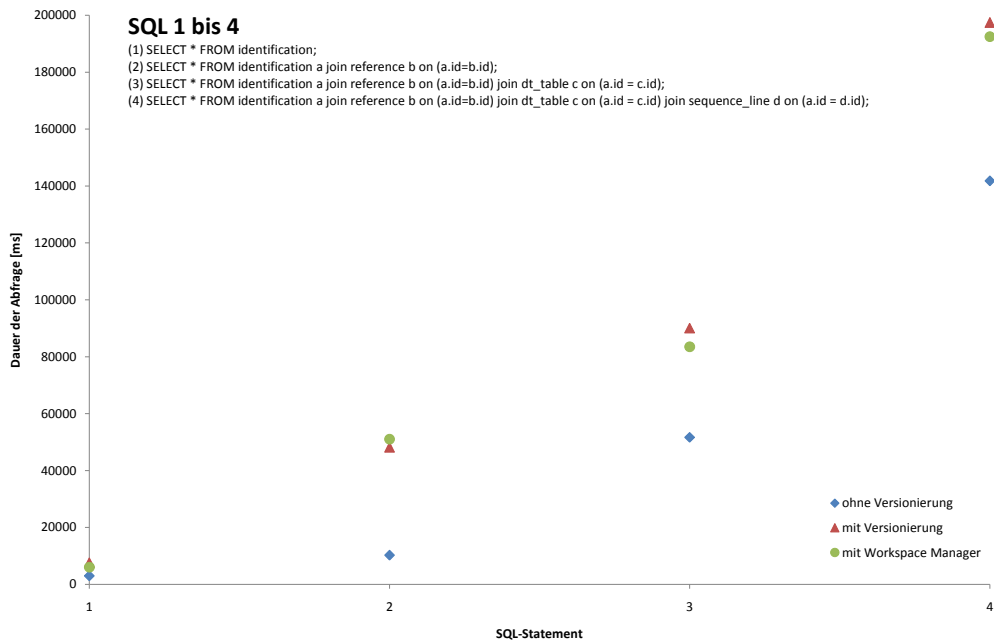


Abbildung 5.3: Dauer der Anfragen SQL 1 bis SQL 4

5.4.2 Anfrage von Bereichen aus Relationen

Um die Zugriffsgeschwindigkeiten in Abhängigkeit von der Selektivität einer Anfrage messen zu können, wurde zuerst bestimmt, mit welchen Sequenzlängen man die Relationen am besten „partitionieren“ kann. In den folgenden Tabellen und Diagrammen kann man die Dauer der durchgeführten Anfragen ersehen.

Auch hier ist wiederum zu erkennen, dass das Einfügen ohne Versionierung am schnellsten gearbeitet hat. Die Werte für die Versionierung und den Oracle Workspace Manager sind weitestgehend ähnlich.

Länge <i>sequence_line</i>	Anzahl Treffer	Selektivität	ohne Versionierung	mit Versionierung	mit Workspace Manager
2	0	0,00%	16 ms	3637 ms	3360 ms
98	15433	10,03%	391 ms	4024 ms	3527 ms
146	30893	20,08%	590 ms	4266 ms	4120 ms
195	46189	30,02%	969 ms	4691 ms	4514 ms
244	61616	40,04%	1156 ms	5028 ms	5198 ms
300	76980	50,03%	1516 ms	5543 ms	5624 ms
354	92182	59,91%	1719 ms	5898 ms	5849 ms
420	107611	69,94%	2016 ms	6391 ms	6190 ms
506	123119	80,01%	2398 ms	6680 ms	6803 ms
691	138497	90,01%	2563 ms	7055 ms	7132 ms
10000	153871	100,00%	2840 ms	7516 ms	7371 ms

Tabelle 5.4: Dauer der Anfrage SQL 5

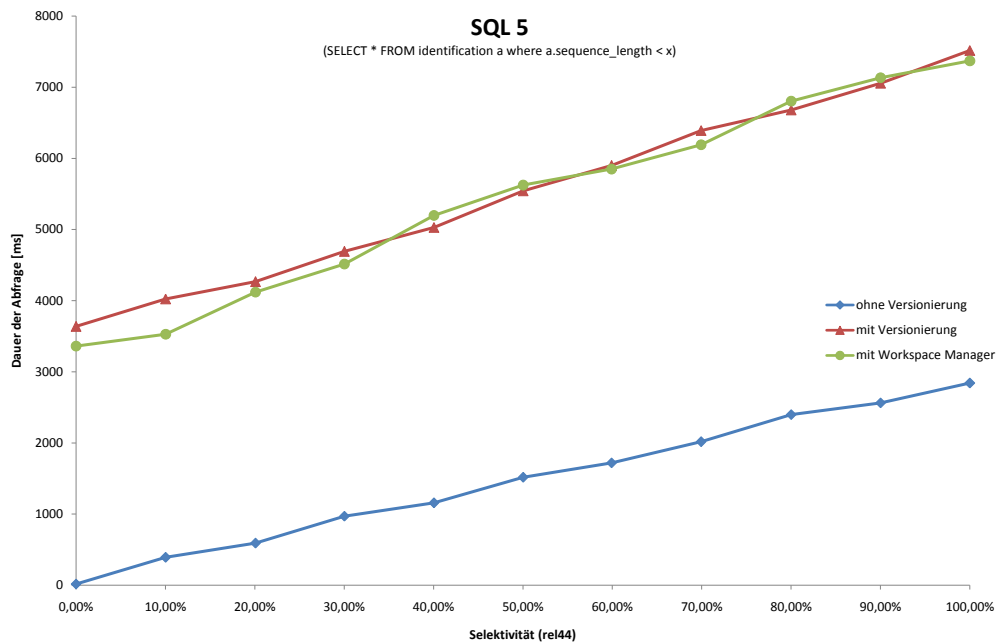


Abbildung 5.4: Dauer der Anfrage SQL 5

Länge <i>sequence_line</i>	Anzahl Treffer	Selektivität	ohne Versionierung	mit Versionierung	mit Workspace Manager
2	0	0,00%	188 ms	3594 ms	3324 ms
98	15433	10,03%	668 ms	7684 ms	6337 ms
146	30893	20,08%	1445 ms	8031 ms	7822 ms
195	46189	30,02%	1422 ms	8547 ms	8193 ms
244	61616	40,04%	1789 ms	8977 ms	9076 ms
300	76980	50,03%	2379 ms	9977 ms	9566 ms
354	92182	59,91%	2547 ms	10106 ms	10067 ms
420	107611	69,94%	2934 ms	10586 ms	10653 ms
506	123119	80,01%	3281 ms	10969 ms	12540 ms
691	138497	90,01%	3766 ms	11414 ms	15438 ms
10000	153871	100,00%	4129 ms	11813 ms	17004 ms

Tabelle 5.5: Dauer der Anfrage SQL 6

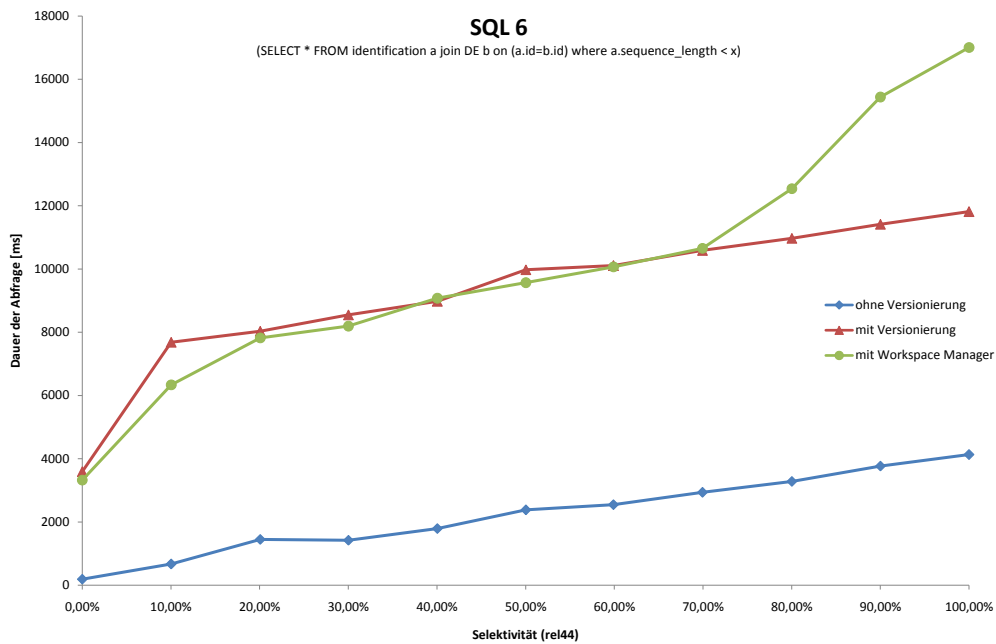


Abbildung 5.5: Dauer der Anfrage SQL 6

Länge <i>sequence_line</i>	Anzahl Treffer	Selektivität	ohne Versionierung	mit Versionierung	mit Workspace Manager
2	0	0,00%	231 ms	10297 ms	10005 ms
98	15433	10,03%	902 ms	15442 ms	13510 ms
146	30893	20,08%	1285 ms	15606 ms	16256 ms
195	46189	30,02%	2961 ms	15770 ms	16495 ms
244	61616	40,04%	2938 ms	16762 ms	16837 ms
300	76980	50,03%	3965 ms	16906 ms	17076 ms
354	92182	59,91%	4559 ms	18176 ms	17755 ms
420	107611	69,94%	6024 ms	18664 ms	18804 ms
506	123119	80,01%	6180 ms	18848 ms	19251 ms
691	138497	90,01%	6586 ms	19121 ms	19429 ms
10000	153871	100,00%	6945 ms	19652 ms	21465 ms

Tabelle 5.6: Dauer der Anfrage SQL 7

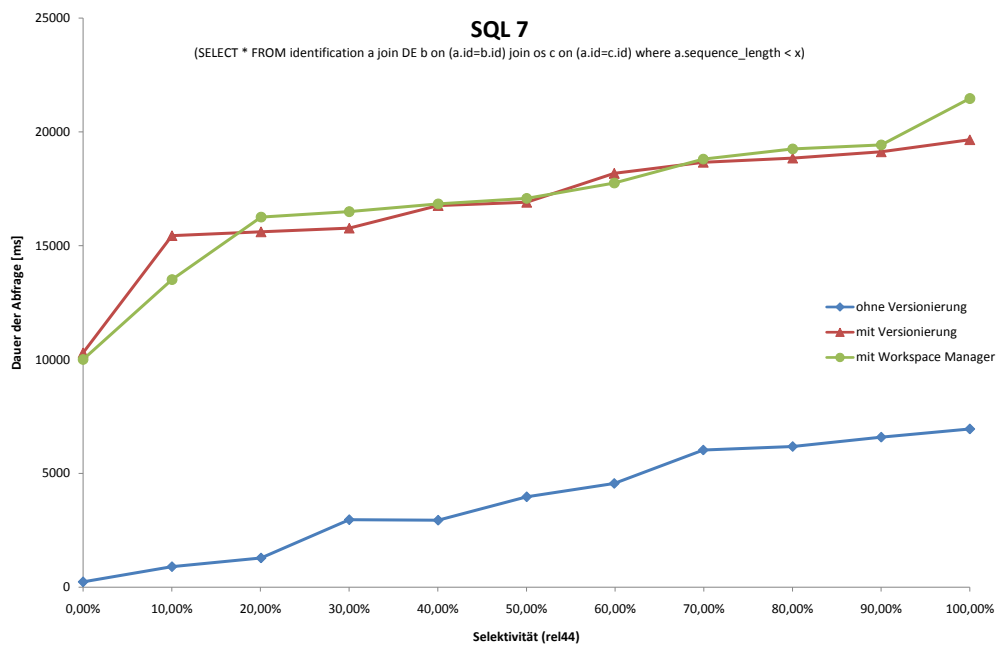


Abbildung 5.6: Dauer der Anfrage SQL 7

Länge <i>sequence_line</i>	Anzahl Treffer	Selektivität	ohne Versionierung	mit Versionierung	mit Workspace Manager
2	0	0,00%	160 ms	15250 ms	14428 ms
98	15433	10,03%	2418 ms	19965 ms	16654 ms
146	30893	20,08%	2879 ms	20059 ms	18428 ms
195	46189	30,02%	4758 ms	20902 ms	20014 ms
244	61616	40,04%	5274 ms	21563 ms	22136 ms
300	76980	50,03%	5840 ms	22070 ms	22185 ms
354	92182	59,91%	6438 ms	22715 ms	22909 ms
420	107611	69,94%	6824 ms	24395 ms	23277 ms
506	123119	80,01%	7426 ms	24520 ms	24332 ms
691	138497	90,01%	8483 ms	25160 ms	26050 ms
10000	153871	100,00%	8816 ms	26254 ms	26671 ms

Tabelle 5.7: Dauer der Anfrage SQL 8

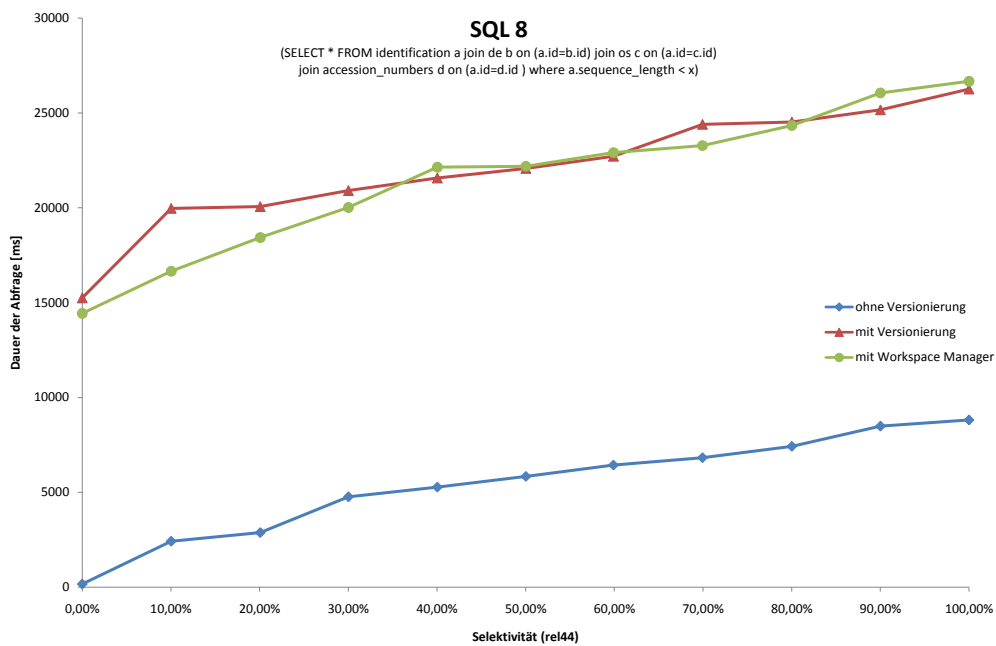


Abbildung 5.7: Dauer der Anfrage SQL 8

Um die Auswirkungen der Versionierung auf die Performance bewerten zu können, wurde die Abfrage SQL8 auch jeweils an den durch Ändern von 10000 Entries (Release 44-1) und zusätzliches Löschen von 5000 Entries (Release 44-2) entstandenen Releases ausgeführt. Die dabei gemessenen Werte sind in den Tabellen 5.8 und 5.9, sowie im Diagramm in der Abbildung 5.8 zu sehen.

Länge <i>sequence_line</i>	Anzahl Treffer	Selektivität	mit Versionierung	mit Workspace Manager
2	0	0,00%	17865 ms	19639 ms
98	15433	10,03%	20037 ms	21588 ms
146	30893	20,08%	22602 ms	23724 ms
195	46189	30,02%	23604 ms	24934 ms
244	61616	40,04%	24360 ms	25708 ms
300	76980	50,03%	25060 ms	26338 ms
354	92182	59,91%	25977 ms	27606 ms
420	107611	69,94%	27306 ms	28841 ms
506	123119	80,01%	28326 ms	30095 ms
691	138497	90,01%	29054 ms	31254 ms
10000	153871	100,00%	30317 ms	32004 ms

Tabelle 5.8: Dauer der Anfrage SQL 8 (Release 44-1)

Länge <i>sequence_line</i>	Anzahl Treffer	Selektivität	mit Versionierung	mit Workspace Manager
2	0	0,00%	19548 ms	21125 ms
98	14933	10,03%	21845 ms	23315 ms
146	29893	20,08%	24843 ms	25545 ms
195	44689	30,02%	25751 ms	26842 ms
244	59616	40,05%	26985 ms	27754 ms
300	74480	50,03%	27845 ms	28422 ms
354	89182	59,90%	28877 ms	29700 ms
420	104111	69,93%	30211 ms	31124 ms
506	119119	80,02%	31124 ms	32333 ms
691	133997	90,01%	31897 ms	33749 ms
10000	148871	100,00%	33245 ms	34613 ms

Tabelle 5.9: Dauer der Anfrage SQL 8 (Release 44-2)

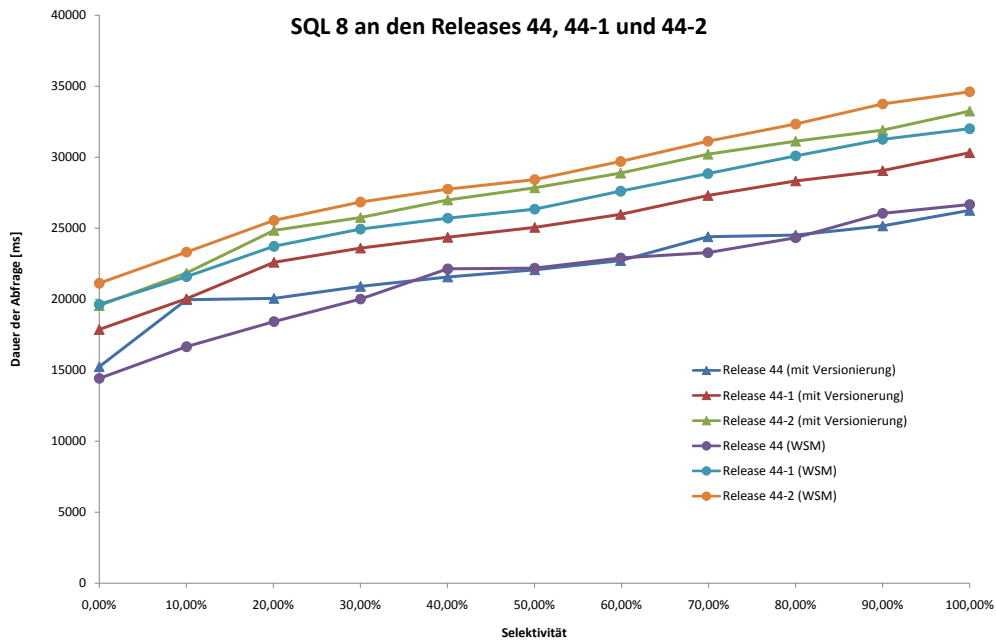


Abbildung 5.8: Dauer der Anfrage SQL 8 (Release 44-1)

Hier ist zu sehen, dass sowohl das Ändern von Daten, wie auch das Löschen, den Zugriff auf die versionierten Releases etwas verlangsamt. Dies ist sowohl in der neu entwickelten Versionierungsstrategie, wie auch im Oracle Workspace-Manager zu erkennen.

5.5 Optimierungen

Zwar wurden mit der neu entwickelten Versionierungsstrategie ähnliche Zugriffszeiten wie beim Oracle Workspace-Manager erreicht, doch war es auch Ziel dieser Studienarbeit, diese Zeiten zu unterbieten. Daher wurden noch einige Optimierungsvarianten getestet.

5.5.1 Alternative Ermittlung der Vorgängerversionen 1

Bisher wurden die Vorgängerversionen mit folgenden Vergleichen ermittelt:

$$versions_pre \leq actual_pre \text{ AND } versions_post \geq actual_post$$

Die Vorgängerversionen können allerdings auch durch

$$versions_pre \leq actual_pre \text{ AND } actual_pre \leq versions_post$$

ermittelt werden. Diese Art der Berechnung der Vorgänger spart somit das Auswerten und Vergleichen mit `actual_pre`.

Die Ergebnisse dieser Optimierungsvariante sind in der Tabelle 5.11 und im Diagramm in der Abbildung 5.9 zu sehen.

Das Ergebnis dieser Optimierung war somit nicht zufrieden stellend.

5.5.2 Alternative Ermittlung der Vorgängerversionen 2

Bei der zweiten Optimierungsvariante im Bereich der Ermittlung der Vorgängerversionen wurde die folgende zusätzliche Tabelle `versionsrel` erzeugt.

version	INTEGER	NOT NULL
parent	INTEGER	NOT NULL

Tabelle 5.10: Tabelle `versionsrel`

In dieser Tabelle wird zu jeder existierenden Version je ein Tupel für ihre Vorgängerversionen eingefügt. Aktualisiert wird diese Tabelle, wenn eine neue Version eingefügt wird. Somit müssen die Vorgängerversionen nicht mehr (aufwändig) über den Pre-Post-Order-Index berechnet werden, sondern können mittels einer einzigen Selectabfrage ausgelesen werden.

5.5.3 Indizierung und Partitionierung

Beim Berechnen der versionierten Releases muss zum einen die Vorgängerversionenkette berechnet werden, zum anderen muss auch in eben dieser Reihenfolge auf die Tupel in der Datentabelle zugegriffen werden. Daher liegt es nahe, die Tabelle `versions` zu indizieren. Zusätzlich ist es sinnvoll auch in der Datentabelle einen Index über `data.version` anzulegen. Um den Zugriff auf die benötigten Teile der Datentabelle zu beschleunigen, wurde diese über `data.version` partitioniert. Somit muss beim Berechnen der versionierten Releases nicht mehrfach die komplette Datentabelle gelesen werden, sondern es werden nur noch die benötigten Partitionen gelesen.

Zusätzlich wurde auch hier die Relation `versionsrel` aus Kapitel 5.5.2 verwendet.

Die Messergebnisse der Optimierung sind im Abschnitt 5.5.4 auf Seite 46 zu finden. Dies ist, wie in den Daten zu sehen ist, die erfolgreichste Optimierungsvariante.

5.5.4 Ergebnisse der Optimierungen

Selektivität	ohne Opt.	mit WSM	Opt. 5.5.1	Opt. 5.5.2	Opt. 5.5.3
0,00%	19548 ms	21125 ms	17200 ms	17082 ms	15274 ms
10,03%	21845 ms	23315 ms	24320 ms	23204 ms	20008 ms
20,08%	24843 ms	25545 ms	24591 ms	23351 ms	20668 ms
30,02%	25751 ms	26842 ms	24619 ms	24213 ms	20965 ms
40,05%	26985 ms	27754 ms	25849 ms	25144 ms	21270 ms
50,03%	27845 ms	28422 ms	26599 ms	26084 ms	22365 ms
59,90%	28877 ms	29700 ms	27036 ms	26375 ms	22621 ms
69,93%	30211 ms	31124 ms	28524 ms	28080 ms	23590 ms
80,02%	31124 ms	32333 ms	30285 ms	28680 ms	23938 ms
90,01%	31897 ms	33749 ms	30695 ms	30307 ms	24785 ms
100,00%	33245 ms	34613 ms	32326 ms	30459 ms	26477 ms

Tabelle 5.11: Dauer der Anfrage SQL 8 (Release 44-2)

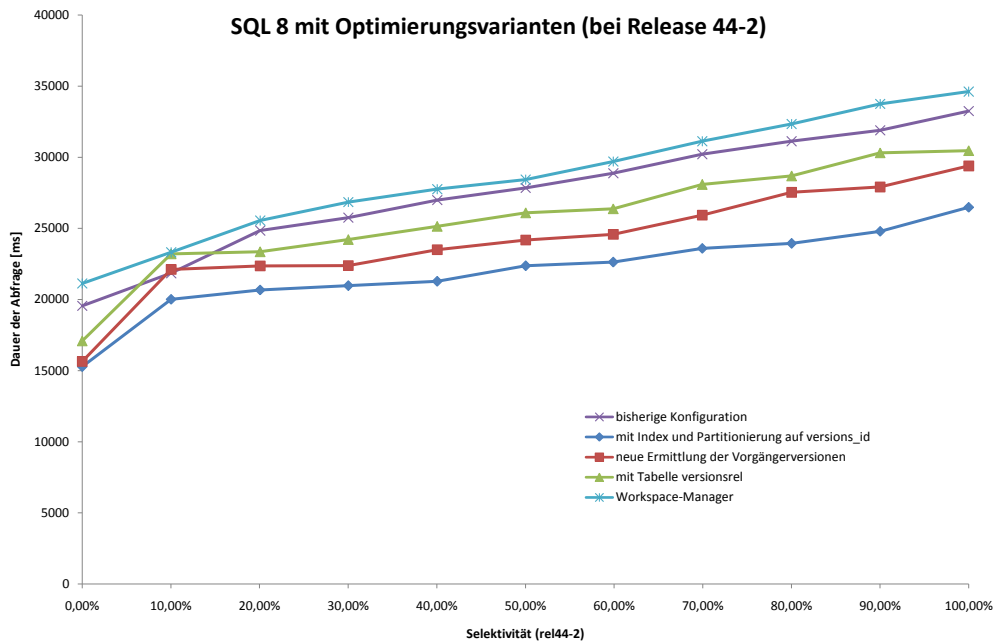


Abbildung 5.9: Dauer der Anfrage SQL 8 (Release 44-2)

Kapitel 6

Fazit

6.1 Constraints

Wichtig zu bemerken ist, dass bei der hier implementierten Versionierung keine automatische Überprüfung von Primary Key und Foreign Key Constraints durchgeführt wird.

Durch das Hinzufügen der zusätzlichen Attribute in die Originaltabellen müssten die bestehenden Primary Keys eigentlich nur um diese Attribute erweitert werden. Dies ist auch möglich. Allerdings kommt es zu Problemen, wenn Fremdschlüssel aus verschiedenen Versionen auf Primärschlüssel in unterschiedlichen Versionen zugreifen. Dies ist insbesondere problematisch, wenn die zu verwendenden Fremdschlüssel auf Primärschlüssel in unterschiedlichen Versionen verweisen.

Eine Möglichkeit besteht darin, die bestehenden Primärschlüssel um die neuen Attribute zu erweitern und damit vom Datenbankmanagementsystem auf Einzigartigkeit prüfen zu lassen. Eine Verwendung von Fremdschlüsselbeziehungen darf dann allerdings nicht erfolgen. Vielmehr muss durch die schreibenden Trigger sichergestellt werden, dass die Fremdschlüsselbeziehungen beachtet werden. Dies gilt nicht nur für das Einfügen und Ändern, sondern auch für das Löschen von Tupeln.

Die um die constraint-Funktionalität erweiterten Trigger waren nicht Inhalt dieser Studienarbeit und wurden daher auch nicht implementiert. Somit sei hier ausdrücklich darauf hingewiesen, dass das implementierte Versionierungssystem keinerlei Unterstützung von constraints anbietet.

6.2 Erweiterungsmöglichkeiten

Eine wichtige Erweiterungsmöglichkeit, die Unterstützung von constraints, insbesondere von Primär- und Fremdschlüsseln, wurde bereits im Abschnitt 6.1 erwähnt.

Weiterhin ist zum Beispiel eine Ausgabe der Baumstruktur des verwendeten Versionenbaumes denkbar.

Momentan müssen die benötigten Trigger und Views noch händig erstellt werden. Hierfür ist der Einsatz eines Programms möglich.

Anhang A

API

Das im Folgenden aufgelistete package *h_versions* stellt alle benötigten Prozeduren und Funktionen der Versionierung zur Verfügung.

Listing A.1: package *h_versions*

```
1 create or replace PACKAGE BODY h_versions AS
2
3     actual_version number := 0;
4     actual_pre number := 0;
5     actual_post number := 0;
6
7     procedure make_child (version_parent INTEGER) AS
8     last_version versions.VERSIONS_ID%type;
9     new_version versions.VERSIONS_ID%type;
10
11     parent_pre versions.VERSIONS_PRE%type;
12     parent_post versions.VERSIONS_POST%type;
13
14     last_version_diff versions.VERSIONS_POST%type;
15     neighbour_post versions.VERSIONS_POST%type;
16
17     new_version_pre versions.VERSIONS_PRE%type;
18     new_version_post versions.VERSIONS_POST%type;
19
20 BEGIN
21     SELECT MAX(versions_id) INTO last_version FROM versions;
22     SELECT versions_pre INTO parent_pre FROM versions
23         WHERE versions_id=version_parent;
24     SELECT versions_post INTO parent_post FROM versions
25         WHERE versions_id=version_parent;
26     SELECT MAX(versions_post) INTO neighbour_post FROM versions
27         WHERE versions_id>version_parent
28         AND versions_post<parent_post
29         AND versions_pre>parent_pre;
30
31     — ID der neuen Version ermitteln
```

```

32     new_version := last_version + 1;
33
34     — PRE und POST der neuen Version ermitteln
35     if (neighbour_post > 0)
36     then
37         last_version_diff := parent_post - neighbour_post;
38         new_version_pre :=
39             trunc(neighbour_post + (last_version_diff / 3));
40         new_version_post :=
41             trunc(neighbour_post + (2 * last_version_diff / 3));
42     else
43         last_version_diff := parent_post - parent_pre;
44         new_version_pre :=
45             trunc(parent_pre + (last_version_diff / 3));
46         new_version_post :=
47             trunc(parent_pre + (2 * last_version_diff / 3));
48     end if;
49
50     if (new_version_pre <= neighbour_post)
51     OR (new_version_pre <= parent_pre)
52     OR (new_version_post >= parent_post)
53     OR (new_version_pre >= new_version_post)
54     then
55         make_tree_bigger(10);
56         make_child(version_parent);
57     else
58         INSERT INTO versions
59             (versions_id , versions_pre , versions_post)
60             VALUES
61             (new_version , new_version_pre , new_version_post);
62         dbms_output.put_line
63             ('Neue Version mit ID ' || versions_id ||
64             ' erstellt. ');
65     end if;
66     commit;
67     END make_child;
68
69     procedure make_tree_bigger (multiplier INTEGER) AS
70     BEGIN
71         UPDATE versions SET versions_pre = versions_pre * multiplier ,
72             versions_post = versions_post * multiplier;
73         commit;
74         dbms_output.put_line
75             ('Der Versionenbaum wurde um de Faktor ' ||
76             multiplier || ' erweitert. ');
77     END make_tree_bigger;
78
79     procedure set_version (versionnr INTEGER) AS
80     BEGIN

```

```

81     actual_version := versionnr;
82     SELECT versions_pre INTO actual_pre FROM versions
83           WHERE versions_id=actual_version;
84     SELECT versions_post INTO actual_post FROM versions
85           WHERE versions_id=actual_version;
86     dbms_output.put_line
87       ('Erfolgreich in Version '||versionnr||' gewechselt. ');
88 END set_version;
89
90
91 function data_alt RETURN data_tabelentyp PIPELINED AS
92 CURSOR prior_versions IS SELECT * FROM data data_1
93     WHERE (
94         data_version IN (
95             SELECT versions_id FROM versions
96             WHERE versions_pre <= actual_pre
97             AND versions_post >= actual_post))
98     AND (NOT EXISTS (
99         SELECT data_id FROM data data_2
100        WHERE ((data_2.data_action=2)
101            AND (data_2.data_version IN (
102                SELECT versions_id FROM versions
103                WHERE versions_pre <= actual_pre
104                AND versions_post >= actual_post))
105            AND (data_2.data_id=data_1.data_id))
106        ))
107     UNION SELECT * FROM data data_1
108     WHERE (
109         (data_action=1)
110         AND (data_version IN (
111             SELECT versions_id FROM versions
112             WHERE versions_pre <= actual_pre
113             AND versions_post >= actual_post
114             ))
115         AND (EXISTS (
116             SELECT data_id FROM data data_2
117             WHERE (
118                 (data_2.data_action=2)
119                 AND (data_2.data_version IN (
120                     SELECT versions_id FROM versions
121                     WHERE versions_pre <= actual_pre
122                     AND versions_post >= actual_post))
123                 AND (data_2.data_id=data_1.data_id)
124                 ))
125             )
126         AND NOT EXISTS (
127             SELECT data_id FROM data data_3
128             WHERE ((data_3.data2_action=2)
129                 AND (data_3.data2_version IN (

```

```

130         SELECT versions_id FROM versions
131         WHERE versions_pre <= actual_pre
132         AND versions_post >= actual_post))
133         AND (data_3.data_id=data_1.data_id)
134         AND (data_3.data_counter > data_1.data2_counter
135         ))));
136
137         prior_versions_data data%ROWTYPE;
138
139 BEGIN
140     OPEN prior_versions;
141     LOOP
142     FETCH prior_versions INTO prior_versions_data;
143
144     IF (prior_versions%FOUND) THEN
145         PIPE ROW (data_spalntyp(prior_versions_data.data_id ,
146         prior_versions_data.data_attr));
147     END IF;
148
149     EXIT WHEN (not prior_versions%FOUND);
150     END LOOP;
151     CLOSE prior_versions;
152     RETURN;
153 END data_alt;
154
155 function next_seqnumber return INTEGER AS
156     next_number INTEGER;
157 BEGIN
158     SELECT v_nummer_seq.nextval INTO next_number FROM dual;
159     return next_number;
160 END next_seqnumber;
161
162 function has_child return INTEGER AS
163     CURSOR children IS SELECT versions_id FROM versions
164     WHERE versions_pre > actual_pre
165     AND versions_post < actual_post;
166     one_child INTEGER;
167     child_exists INTEGER;
168 BEGIN
169     child_exists := 0;
170     OPEN children;
171     LOOP
172     FETCH children INTO one_child;
173     IF (children%FOUND)
174     THEN child_exists := 1;
175     END IF;
176     EXIT WHEN (not children%FOUND);
177     END LOOP;
178     CLOSE children;

```

```

179     return child_exists;
180 END has_child;
181
182 function show_last_version return INTEGER AS
183     last_version INTEGER;
184 BEGIN
185     SELECT MAX(versions_id) INTO last_version FROM versions;
186     dbms_output.put_line('Letzte Versions-ID: '||last_version);
187     return lastversion;
188 END show_last_version;
189
190 BEGIN
191     set_version(0);
192 END h_versions;

```

Anhang B

Algorithmus der Messung

Für die Messung wurde folgendes Skript benutzt:

Listing B.1: Algorithmus der Messung

```
1  SET SERVEROUTPUT ON;
2  DECLARE
3      ts1 TIMESTAMP;
4      ts2 TIMESTAMP;
5      CURSOR retvalcurs is SELECT * FROM IDENTIFICATION;
6      retval retvalcurs%ROWTYPE;
7
8  BEGIN
9      --erstmal ohne Buffer
10     SELECT SYSTIMESTAMP INTO ts1 FROM DUAL;
11     OPEN retvalcurs;
12     LOOP
13         FETCH retvalcurs INTO retval;
14         EXIT WHEN (not retvalcurs%FOUND);
15     END LOOP;
16     CLOSE retvalcurs;
17     SELECT SYSTIMESTAMP INTO ts2 FROM DUAL;
18     DBMS_OUTPUT.PUT_LINE('Zeit_(ohne_Buffer):_');
19     DBMS_OUTPUT.PUT_LINE(TO_CHAR((ts2-ts1),
20         'YYYY-MM-DD_HH:MI:SSXFF'));
21
22     --mit Buffer
23     SELECT SYSTIMESTAMP INTO ts1 FROM DUAL;
24     OPEN retvalcurs;
25     LOOP
26         FETCH retvalcurs INTO retval;
27         EXIT WHEN (not retvalcurs%FOUND);
28     END LOOP;
29     CLOSE retvalcurs;
30     SELECT SYSTIMESTAMP INTO ts2 FROM DUAL;
31     DBMS_OUTPUT.PUT_LINE('Zeit_(mit_Buffer):_');
32     DBMS_OUTPUT.PUT_LINE(TO_CHAR((ts2-ts1),
```

```

33         'YYYY-MM-DD_HH:MI:SSXFF' ));
34
35 SELECT SYSTIMESTAMP INTO ts1 FROM DUAL;
36 OPEN retvalcurs;
37     LOOP
38         FETCH retvalcurs INTO retval;
39         EXIT WHEN (not retvalcurs%FOUND);
40     END LOOP;
41 CLOSE retvalcurs;
42 SELECT SYSTIMESTAMP INTO ts2 FROM DUAL;
43 DBMS.OUTPUT.PUT_LINE('Zeit_(mit_Buffer):_');
44 DBMS.OUTPUT.PUT_LINE(TO.CHAR((ts2-ts1),
45     'YYYY-MM-DD_HH:MI:SSXFF' ));
46
47 SELECT SYSTIMESTAMP INTO ts1 FROM DUAL;
48 OPEN retvalcurs;
49     LOOP
50         FETCH retvalcurs INTO retval;
51         EXIT WHEN (not retvalcurs%FOUND);
52     END LOOP;
53 CLOSE retvalcurs;
54 SELECT SYSTIMESTAMP INTO ts2 FROM DUAL;
55 DBMS.OUTPUT.PUT_LINE('Zeit_(mit_Buffer):_');
56 DBMS.OUTPUT.PUT_LINE(TO.CHAR((ts2-ts1),
57     'YYYY-MM-DD_HH:MI:SSXFF' ));
58 END;

```

Die entsprechenden SQL-Abfragen SQL 1 bis SQL 8 wurden jeweils in Zeile 5 eingefügt.

Anhang C

Struktur von SwissProt

Listing C.1: Struktur von *SwissProt*

```
1  CREATE TABLE IDENTIFICATION(  
2      ID CHAR(10) PRIMARY KEY,  
3      ENTRY_NAME CHAR(10) ,  
4      DATA_CLASS CHAR(11) ,  
5      MOLECULE_TYPE CHAR(3) ,  
6      SEQUENCE_LENGTH INTEGER  
7  );  
8  
9  CREATE TABLE DT_TABLE(  
10     ID CHAR(10) REFERENCES IDENTIFICATION  
11         ON DELETE CASCADE,  
12     N INTEGER,  
13     DT DATE,  
14     COM CHAR(40) ,  
15     PRIMARY KEY(ID ,N)  
16 );  
17  
18 CREATE TABLE ACCESSION_NUMBERS(  
19     ID CHAR(10) REFERENCES IDENTIFICATION  
20         ON DELETE CASCADE,  
21     N INTEGER,  
22     ACCESSION_NUMBER CHAR(10) ,  
23     PRIMARY KEY(ID ,N)  
24 );  
25  
26 CREATE TABLE DE(  
27     ID CHAR(10) REFERENCES IDENTIFICATION  
28         ON DELETE CASCADE,  
29     DESCRIPTION VARCHAR2(2000 CHAR) ,  
30     PRIMARY KEY(ID)  
31 );  
32  
33  
34 CREATE TABLE OS(  
35
```

```

35     ID CHAR(10) REFERENCES IDENTIFICATION
36     ON DELETE CASCADE,
37     N INTEGER,
38     ORGANISM_SPECIES CHAR(100) ,
39     PRIMARY KEY(ID ,N)
40 );
41
42 CREATE TABLE OG(
43     ID CHAR(10) REFERENCES IDENTIFICATION
44     ON DELETE CASCADE,
45     N INTEGER,
46     ORGANELLE CHAR(100) ,
47     PRIMARY KEY(ID ,N)
48 );
49
50 CREATE TABLE OC(
51     ID CHAR(10) REFERENCES IDENTIFICATION
52     ON DELETE CASCADE,
53     ORGANISM_CLASSIFICATION VARCHAR2(1000) ,
54     PRIMARY KEY(ID)
55 );
56
57 CREATE TABLE REFERENCE(
58     ID CHAR(10) REFERENCES IDENTIFICATION
59     ON DELETE CASCADE,
60     RN INTEGER,
61     RP CHAR(100) ,
62     RX CHAR(100) ,
63     RA VARCHAR2(4000 CHAR) ,
64     RL VARCHAR2(2000 CHAR) ,
65     PRIMARY KEY(ID ,RN)
66 );
67
68 CREATE TABLE REFERENCE.COMMENT(
69     ID CHAR(10) ,
70     RN INTEGER,
71     TEXT VARCHAR(1000) ,
72     PRIMARY KEY(ID ,RN) ,
73     FOREIGN KEY(ID ,RN) REFERENCES REFERENCE
74     ON DELETE CASCADE
75 );
76
77 CREATE TABLE SEQUENCE_LINE(
78     ID CHAR(10) REFERENCES IDENTIFICATION
79     ON DELETE CASCADE,
80     MOLECULAR_WEIGHT INTEGER,
81     CONTROLLNUMBER CHAR(30) ,
82     SEQUENCE_DATA CLOB,
83     PRIMARY KEY(ID)

```

```
84 );  
85  
86 CREATE TABLE COMMENTS(  
87     ID CHAR(10) REFERENCES IDENTIFICATION  
88     ON DELETE CASCADE,  
89     TEXT CLOB,  
90     PRIMARY KEY(ID)  
91 );
```


Literaturverzeichnis

- [1] Rieche, St.: Diplomarbeit „Versionierung in relationalen Datenbanken“, 2004
- [2] Türscher, G.: PL/SQL – Lernen, Verstehen und Einsetzen, Springer 1997
- [3] Raymans, H.-G.: ORACLE Programmierung, Addison–Wesley 2001
- [4] Feuerstein, St., Pribyl, B.: Oracle PL/SQL, O´Reilly 1999
- [5] Abrahamson, I., Abbey, M., Corey, M.: Oracle Database 10g für Einsteiger, Oracle Press 2004
- [6] Loney, K., Koch, G.: Oracle 9i Die umfassende Referenz, Oracle Press 2003
- [7] Zhu, N.: Data Versioning Systems, Stony Brook, 2003
- [8] Klahold, P., Schlageter, G., Wilkes, W.: A General Model for Version Management in Databases, Proceedings of 12th International Conference on Very Large Databases, 1986
- [9] Bebel, B., Eder, J., Koncilia, Ch., Morzy, T., Wrembel, R.: Creation and Management of Versions in Multiversion Data Warehouse, ACM Symposium on Applied Computing, 2004
- [10] Swiss Institute of Bioinformatics - University of Geneva, EMBL Outstation, EBI: Swiss-Prot, Protein knowledgebase., URL: <http://www.expasy.org/sprot/>

Abbildungsverzeichnis

2.1	Lineare Versionierung	11
2.2	Hierarchische Versionierung	11
3.1	Pre-Post-Order-Index	18
4.1	Versionenbaum	30
5.1	Geschwindigkeit des Einfügens des Release 37	35
5.2	Geschwindigkeit des Einfügens des Release 44	36
5.3	Dauer der Anfragen SQL 1 bis SQL 4	38
5.4	Dauer der Anfrage SQL 5	39
5.5	Dauer der Anfrage SQL 6	40
5.6	Dauer der Anfrage SQL 7	41
5.7	Dauer der Anfrage SQL 8	42
5.8	Dauer der Anfrage SQL 8 (Release 44-1)	44
5.9	Dauer der Anfrage SQL 8 (Release 44-2)	46

Tabellenverzeichnis

2.1	Tabelle <i>A</i> vor der Versionierung	13
2.2	Tabelle <i>A</i> nach dem Einfügen der Versionierungsattribute	13
2.3	Tabelle <i>A</i> vor der Versionierung	14
2.4	Teil-Tabelle <i>A_1</i> nach der Zerlegung	15
2.5	Teil-Tabelle <i>A_2</i> nach der Zerlegung	15
3.1	Tabelle <i>versions</i>	19
3.2	Tabelle <i>data</i> (alt)	20
3.3	Tabelle <i>data</i> (neu)	21
3.4	Initialwerte	21
4.1	Struktur der Tabelle <i>data</i>	25
4.2	Inhalt der Tabelle <i>data</i>	25
4.3	Struktur der Tabelle <i>hv_data</i>	26
4.4	Struktur der Tabelle <i>versions</i>	26
4.5	Inhalt der Tabelle <i>hv_data</i>	26
4.6	Inhalt der Tabelle <i>versions</i>	26
4.7	neue Tupel für der Tabelle <i>data</i>	27
4.8	Inhalt der Tabelle <i>hv_data</i>	28
4.9	Inhalt der Tabelle <i>versions</i>	28
4.10	Anlegen der Versionen 1 bis 3 unter der Version 0	28
4.11	Erläuterung der Spaltennamen von Tabelle 4.10	29
4.12	neue Tupel für der Tabelle <i>data</i>	29
4.13	Inhalt der Tabelle <i>hv_data</i>	29
4.14	Inhalt der Tabelle <i>versions</i>	30
4.15	Inhalt der Tabelle <i>data</i> – Version 0	30
4.16	Inhalt der Tabelle <i>data</i> – Version 1	30
4.17	Inhalt der Tabelle <i>data</i> – Version 2	31

4.18	Inhalt der Tabelle <i>data</i> – Version 3	31
4.19	Inhalt der Tabelle <i>data</i> – Version 4	31
4.20	Inhalt der Tabelle <i>data</i> – Version 5	31
5.1	Entries pro Sekunde beim Einfügen	34
5.2	Dauer des Einfügens Einfügen in hh:mm:ss	34
5.3	Dauer der Anfragen SQL 1 bis SQL 4	37
5.4	Dauer der Anfrage SQL 5	39
5.5	Dauer der Anfrage SQL 6	40
5.6	Dauer der Anfrage SQL 7	41
5.7	Dauer der Anfrage SQL 8	42
5.8	Dauer der Anfrage SQL 8 (Release 44-1)	43
5.9	Dauer der Anfrage SQL 8 (Release 44-2)	43
5.10	Tabelle <i>versionsrel</i>	45
5.11	Dauer der Anfrage SQL 8 (Release 44-2)	46

