

ENTWICKLUNG EINES METADATENGESTEUERTEN HISTORISIERUNGSWERKZEUGS FÜR DIE AKTUALISIERUNG VON DATA WAREHOUSES

Diplomarbeit im Fach Informatik

Vorgelegt von

STEPHANIE BALZER
RAFZ, SCHWEIZ
MATRIKELNUMMER 94-723-046

Angefertigt am
Institut für Informatik der Universität Zürich

Prof. Dr. Klaus R. Dittrich

Betreuung: Dr. Anca Vaduva
Abgabe der Arbeit: 28. November 2001

VORWORT

Die vorliegende Arbeit ist als Informatik-Diplomarbeit in der Datenbanktechnologie-Gruppe der Universität Zürich entstanden und bildet den Schlusspunkt meines Wirtschaftsinformatikstudiums.

Es ist mir ein Anliegen, den folgenden Personen für ihren unschätzbaren Beitrag zum Gelingen dieser Arbeit zu danken:

Als Leiter der Datenbanktechnologie-Gruppe danke ich Herrn Prof. Dr. Klaus R. Dittrich herzlich, dass er mir die Durchführung dieser interessanten Arbeit ermöglichte. Ich empfand das Arbeitsklima in seiner Gruppe als äusserst motivierend und habe mich während meiner ganzen Diplomarbeit begleitet gefühlt.

Ebenso geht mein herzlicher Dank an meine Betreuerin Frau Dr. Anca Vaduva, wissenschaftliche Mitarbeiterin der Datenbanktechnologie-Gruppe. Sie ist mir während der ganzen Zeit des Erarbeitens mit ihrem fachkundigen Rat zur Seite gestanden und hat mich jederzeit vollumfänglich unterstützt. Besonders geschätzt habe ich, dass ich mich bei ihr nicht nur fachlich, sondern auch menschlich vollauf aufgehoben fühlen durfte.

Nicht zuletzt möchte ich auch meine Familie und meinen Partner Adnan erwähnen, die für mich „mein Zuhause“ sind.

INHALTSVERZEICHNIS

1	EINLEITUNG	1
2	EINFÜHRUNG INS DATA WAREHOUSING	3
2.1	OPERATIVE UND ANALYTISCHE DATENBANKEN.....	3
2.1.1	<i>Operative versus analytische Datenbanken</i>	<i>3</i>
2.1.2	<i>Data Warehouse und Data Warehousing.....</i>	<i>5</i>
2.2	ARCHITEKTUR EINES DATA-WAREHOUSE-SYSTEMS	5
2.2.1	<i>Akquisition.....</i>	<i>6</i>
	Extraktion.....	7
	Transformation und Integration	7
	Bereinigung	7
	Vervollständigung	7
	Historisierung.....	8
	Laden.....	8
2.2.2	<i>Modellierung und Speicherung.....</i>	<i>8</i>
2.2.3	<i>Zugriff.....</i>	<i>9</i>
2.3	MULTIDIMENSIONALE DATENMODELLIERUNG	9
2.3.1	<i>Multidimensionales Datenmodell.....</i>	<i>9</i>
2.3.2	<i>Speicherung multidimensionaler Daten</i>	<i>11</i>
	Star-Schema	11
2.4	METADATEN IM DATA WAREHOUSING	12
3	HISTORISIERUNG	14
3.1	HISTORISIERUNGSASPEKTE IN TEMPORALEN DATENBANKEN	14
3.1.1	<i>Zeitrepräsentation</i>	<i>15</i>
3.1.2	<i>Gültigkeitszeit versus Transaktionszeit</i>	<i>15</i>
3.1.3	<i>Momentaufnahme versus Zustandsdauer</i>	<i>17</i>
3.1.4	<i>Tupel-Versionierung versus Attribut-Versionierung.....</i>	<i>19</i>
3.2	TEMPORALE ERWEITERUNGEN.....	20
3.2.1	<i>Temporale Erweiterung einer relationalen Datenbank</i>	<i>20</i>
	Gültigkeitszeit mit Zeitintervallen	22
	Gültigkeitszeit mit Momentaufnahmen.....	25
	Gültigkeits- und Transaktionszeit mit Zeitintervallen.....	27
	Gültigkeits- und Transaktionszeit mit Momentaufnahmen.....	31
	Referentielle Integrität.....	32
3.2.2	<i>Temporale Erweiterung einer Datenbank mit komplexen Strukturen.....</i>	<i>33</i>
3.3	HISTORISIERUNGSASPEKTE IM MULTIDIMENSIONALEN DATENMODELL	34
3.3.1	<i>Herkömmliches Star-Schema</i>	<i>34</i>
	Historisierung von Fakten	34
	Historisierung von Dimensionen.....	35
3.3.2	<i>Temporales Star-Schema.....</i>	<i>37</i>
3.4	HISTORISIERUNG VON METADATEN.....	39
3.5	ABSTRAKTES HISTORISIERUNGSMODELL	39
3.5.1	<i>Dimensionen der Historisierung.....</i>	<i>39</i>
	Zeitangabe	40
	Zeitgranulat	40
	Zeitdimensionen	40
	Datengranulat	41
	Entwicklungsbeschreibung.....	41

Entwicklungsprozess.....	41
Wachstumsrichtung.....	42
3.5.2 <i>Anwendung des abstrakten Historisierungsmodells</i>	42
4 METADATENGESTEUERTE HISTORISIERUNG IM DATA WAREHOUSING.....	44
4.1 EINBETTUNG INS DATA-WAREHOUSE-SYSTEM.....	45
4.2 METADATENSHEMA.....	45
4.3 ARCHITEKTUR.....	48
5 PROTOTYP.....	49
5.1 BESCHREIBUNG.....	49
5.1.1 <i>Architektur und Systemangaben</i>	49
5.1.2 <i>Java-Applikation</i>	50
5.1.3 <i>Testumgebung</i>	53
5.2 ERKENNTNISSE.....	54
6 ZUSAMMENFASSUNG.....	58
ANHANG A: QUELLTEXT ZUM PROTOTYP.....	60
LITERATURVERZEICHNIS.....	95
ABBILDUNGSVERZEICHNIS.....	97
TABELLENVERZEICHNIS.....	98

1 EINLEITUNG

Wie der Titel der vorliegenden Arbeit verrät, lassen sich diese Seiten dem weiten Feld der Datenbanktechnologie und dort dem Spezialgebiet Data Warehousing zuordnen.

Mit dem Anwachsen des tertiären Sektors und dem Wandel hin zu einer Informationsgesellschaft wurde Information zu einem wichtigen Produktionsfaktor. Information an sich ist aber wertlos, wenn sie nicht zugänglich gemacht und entsprechend verwaltet wird. Diesem Anliegen nimmt sich das Gebiet der Datenbanktechnologie an, indem **Datenbanksysteme** die dauerhafte und zuverlässige Verwaltung von elektronischen Daten zum schnellen Zugriff durch mehrere, gleichzeitig anfragende Benutzer erlauben.

Datenbanksysteme unterstützen heute zahlreiche Geschäftsprozesse, indem sie zur Erledigung operativer Aufgaben eingesetzt werden. Viele Dienstleistungsanbieter, wie etwa Banken und Versicherungen, könnten heute sogar ohne Datenbanksysteme schlichtweg nicht mehr existieren. Im Verlauf der letzten Jahre hat sich aber neben der Unterstützung des operativen Geschäfts ein weiterer Anwendungsbereich für Datenbanken herauskristallisiert: die Entscheidungsunterstützung. Dahinter steht die Absicht, die in operativen Datenbanken zahlreich, oft auch implizit enthaltene Information für Analysezwecke zu verwenden und dadurch eine Entscheidungsgrundlage zu schaffen. Weil aber die meisten Unternehmungen über mehrere operative Datenbanksysteme verfügen, die zudem oft noch unterschiedlich sind, ist ein einheitlicher Zugriff auf alle Quellen meist verwehrt. Hier kann ein Data Warehouse Abhilfe leisten. Dabei kann man sich unter einem **Data Warehouse** eine Datensammlung vorstellen, welche Daten aus mehreren operativen oder sonstigen externen Quellen in sich vereint und dadurch einen einheitlichen Zugriff auf die Daten zur Entscheidungsfindung erlaubt. Selbstverständlich braucht es zur Verwaltung einer solchen Datenbasis ein entsprechendes Datenhaltungssystem. Ein weiterer Grund für den Einsatz von Data Warehouses ist die Absicht, die operativen Datenquellen zu entlasten, indem diese nicht auch noch Anfragen zur Entscheidungsunterstützung befriedigen müssen. Data Warehouses erfreuen sich heutzutage einer grossen Popularität.

Neben weiteren wichtigen Eigenschaften eines Data Warehouse ist die Aufrechterhaltung der Historie der in einem Data Warehouse gespeicherten Daten – also deren **Historisierung** – ein zentrales Anliegen. Dahinter steht die Erkenntnis, dass es zur Entscheidungsunterstützung nicht nur die aktuellen, sondern auch die vergangenen Daten braucht, weshalb bei Änderungen alte Werte nicht überschrieben, sondern beibehalten werden.

Ein weiterer Forschungsbereich befasst sich mit der Rolle von **Metadaten** im Data Warehousing. Dabei beschreiben Metadaten als „Daten über Daten“ die Struktur der in einem Data Warehouse gespeicherten Daten. Es zeigt sich, dass Metadaten eine bedeutende Rolle im Data Warehousing innehaben und deshalb zusätzlich zu den eigentlichen Daten verwaltet werden sollten.

Nach diesen Ausführungen lässt sich die dieser Arbeit zugrunde liegende Aufgabenstellung erläutern: Ziel der Arbeit ist, ein metadatengesteuertes Historisierungswerkzeug für die Aktualisierung von Data Warehouses zu entwickeln. Um dies zu erreichen, gilt es, sich in einem ersten Schritt in die Historisierung im Allgemeinen und im Speziellen bei Data Warehouses einzuarbeiten und dadurch Historisierungskonzepte zu gewinnen. Dann muss die Rolle von Metadaten für die Historisierung diskutiert und untersucht werden, welche Metadaten für die Historisierung relevant sind. Erst nach dieser geleisteten Vorarbeit kann die Entwicklung des Historisierungswerkzeugs beginnen. Dieses Historisierungswerkzeug soll – wie sein Name es verrät – ins Data Warehouse einzubringende Daten übertragen und dabei die Historie der Daten aufrecht erhalten. Eine weitere Anforderung an das Werkzeug ist seine **Metadatensteuerung**. Dahinter steht die Idee, die für die Historisierung benötigten Metadaten nicht versteckt im Historisierungswerkzeug zu halten, sondern öffentlich zugänglich in einem Repository abzuspeichern. Erst zur Laufzeit werden solche Steuerungsinformationen eingelesen und in den Programmtext eingebunden. Diese Vorgehensweise

zeichnet sich vor allem durch eine erhöhte Flexibilität und Wiederverwendbarkeit aus. Schliesslich darf sich der Einfachheit halber das zu entwickelnde Werkzeug auf den relationalen Fall beschränken.

Da es zur Zeit noch wenige, nur isolierte Erkenntnisse bezüglich der Historisierung, geschweige denn ein einheitliches Rahmenmodell dafür gibt, wird sich die vorliegende Arbeit in einem ersten Schritt auf die Erarbeitung eines solchen Rahmenmodells konzentrieren müssen. Dabei soll die Wahrung einer gewissen Universalität, also die uneingeschränkte Anwendbarkeit des Modells, im Vordergrund stehen. Sollte ein solches Rahmenmodell gefunden werden, wäre ein vielversprechender Beitrag geleistet: Zum einen liessen sich die bisherigen Erkenntnisse unter einem einheitlichen Blickwinkel vereinen und würden dadurch auch miteinander vergleichbar, zum anderen würde ein solches Rahmenmodell die Entwicklung eines Historisierungswerkzeugs vereinfachen, indem sich die Implementierung gewissermassen am Modell ausrichten kann. Zudem könnte ein solches Historisierungsmodell auch die Diskussion über die Rolle von Metadaten für die Historisierung massgeblich bestimmen und dadurch die Implementierung eines metadatengesteuerten Werkzeugs vereinfachen. Insofern liessen sich neue Erkenntnisse über den Einsatz metadatengesteuerter Werkzeuge im Data Warehousing gewinnen.

Die vorliegende Arbeit setzt grundlegende Kenntnisse in der Datenbanktechnik voraus. Dazu zählen insbesondere das relationale, objektrelationale und objektorientierte Datenmodell [ElNa], [Codd], [Catt]. Zudem erfordert die Implementierung Kenntnisse in der objektorientierten Modellierung und in der objektorientierten Programmiersprache Java. Weitere für diese Arbeit benötigte Konzepte, wie etwa Data Warehousing, werden im Verlauf ausführlich behandelt.

Die verbleibenden Seiten dieser Arbeit sind folgendermassen strukturiert: Kapitel 2 gibt eine Einführung ins Data Warehousing und schafft dadurch die Grundlage für die Erörterung der in dieser Arbeit zu behandelnden Fragestellung. In Kapitel 3 wird ein konzeptionelles Rahmenmodell für die Historisierung, ein so genanntes abstraktes Historisierungsmodell, erarbeitet. Dieses bildet fortan die Basis, an welcher sich die restlichen Kapitel ausrichten. Während sich Kapitel 4 auf einer konzeptionellen Ebene mit der metadatengesteuerten Historisierung im Data Warehousing auseinandersetzt, stellt Kapitel 5 den in dieser Arbeit entwickelten Prototyp als konkrete Implementierung für ein metadatengesteuertes Werkzeug vor. Der Quelltext ist dabei dem Anhang beigelegt. Kapitel 6 schliesslich bildet als Zusammenfassung den Abschluss der vorliegenden Arbeit.

2 EINFÜHRUNG INS DATA WAREHOUSING

In diesem Kapitel wird in das Gebiet Data Warehousing eingeführt, um einerseits ein grundlegendes Verständnis für die Thematik zu schaffen und andererseits den Weg für die in dieser Arbeit diskutierten Fragestellung zu ebnet.

Auch im Gebiet Data Warehousing ist man leider nicht vor dem in der Informatik allzu oft verbreiteten Übel gefeit, dass die gleichen Begriffe für Unterschiedliches verwendet werden. Um von Anfang an Klarheit zu verschaffen, sei festgehalten, dass sich die vorliegende Arbeit weitgehend an der Terminologie in [Ditt] orientiert.

Im folgenden Abschnitt 2.1 wird nebst einer historischen Einbettung auch die Abgrenzung zwischen so genannten operativen und analytischen Datenbanken vollzogen. Zudem werden die Begriffe Data Warehouse und Data Warehousing erläutert. Abschnitt 2.2 zeigt in schematischer Weise die grundlegende Architektur eines Data-Warehouse-Systems und Abschnitt 2.3 diskutiert die in Data Warehouses weit verbreitete multidimensionale Datenmodellierung. Abschliessend wird in Abschnitt 2.4 die Rolle von Metadaten im Data Warehousing eingehender untersucht.

2.1 OPERATIVE UND ANALYTISCHE DATENBANKEN

Mit dem Aufkommen der elektronischen Datenverarbeitung wurde es erstmals möglich, **Geschäftsprozesse** durch Informationssysteme zu **unterstützen**. Da in den meisten Geschäftsprozessen eine grosse Menge an Daten anfallen, die es dauerhaft zu verwalten gilt, kommen dafür Datenbankverwaltungssysteme zum Einsatz. Aufgrund ihrer Unterstützung in der täglichen Führung eines Unternehmens werden solche Datenbankverwaltungssysteme als **operative Datenbankverwaltungssysteme** bzw. **operative Datenbanken** bezeichnet. Häufig findet man auch den Begriff OLTP-Systeme vor. **OLTP** steht für **Online Transaction Processing** und verdeutlicht den Aspekt, dass solche Informationssysteme auf einen schnellen Zugriff hin und für eine Massentransaktionsverarbeitung optimiert sind.

Durch die weite Verbreitung und den täglichen Einsatz ist enorm viel Information in operativen Datenbanken enthalten. Nebst den Informationen, die für das tägliche Geschäft benötigt werden, finden sich darin noch weitere, meist implizite Informationen, die für die Unternehmensführung von höchster Relevanz sind. Es gilt also, diese Informationen zur **Entscheidungsunterstützung** zugänglich zu machen. Datenbankverwaltungssysteme, die den Zugriff und die Analyse von Daten zur Entscheidungsunterstützung möglich machen, werden als **analytische Datenbankverwaltungssysteme** bzw. **analytische Datenbanken** bezeichnet. Selbstverständlich muss ein solches Datenbanksystem die gängigen Anforderungen an ein Datenbanksystem wie Persistenz, Datenunabhängigkeit, Mehrfachbenutzbarkeit usw. erfüllen. Zuweilen wird auch der Begriff **DSS**, **Decision Support System**, für analytische Datenbanksysteme verwendet, um den entscheidungsunterstützenden Charakter zu unterstreichen.

2.1.1 Operative versus analytische Datenbanken

Wie bereits einleitend erwähnt, hat sich mit der Zeit neben dem Ruf nach Unterstützung des operativen Geschäfts auch derjenige nach einer **Entscheidungsunterstützung** breit gemacht. Es lassen sich somit operative von analytischen Datenbanken abgrenzen. Operative und analytische Datenbanken unterscheiden sich aber nicht nur hinsichtlich ihres Einsatzes, es lassen sich weitere Kriterien aufführen, die die Komplementarität beider Datenbanken verdeutlichen. Dies geht aus Tabelle 2-1 hervor.

	operative Datenbank	analytische Datenbank
Einsatz/Anwendung	operatives Geschäft	Analyse, Entscheidungsunterstützung
Daten	aktuell, isoliert, detailliert	historisiert, integriert, detailliert und aggregiert
Verarbeitung	Transaktionen, häufige Änderungen	Anfragen, Ergänzungen
Eigenschaften	Konsistenz, Vollständigkeit	Qualität, Richtigkeit
Entwurf	einheitenorientiert, redundanzarm, normalisiert	themenorientiert, nur teilweise normalisiert
Grösse	MB bis GB	GB bis TB

Tabelle 2-1: Operative versus analytische Datenbanken, Quelle: [Ditt]

Abgesehen vom ersten Punkt aus Tabelle 2-1, der bereits eingehend ausgeleuchtet wurde, werden die restlichen Unterscheidungskriterien nachfolgend kurz erläutert:

- ❖ **Daten:** Während bei operativen Datenbanken die Datenbasis zu einem Zeitpunkt lediglich den **aktuellen** Zustand widerspiegelt, werden in analytischen Datenbanken **historisierte** Daten gehalten (siehe Kapitel 3). Dies bedeutet, dass die zeitliche Entwicklungsgeschichte der Daten ersichtlich wird: Alte Werte werden nicht - wie bei operativen Datenbanken üblich - überschrieben, sondern beibehalten. Die neuen Werte gehen als zusätzliche Einträge in den Datenbestand ein. Es zeigt sich somit, dass in analytischen Datenbanken die Zeit und die **zeitbezogene Analyse** eine fundamentale Rolle spielt.

Die zweite Eigenschaft macht deutlich, dass in analytischen Datenbanken Daten verschiedenster, unter Umständen heterogener Quellen zur Analyse beigezogen werden, wobei eine **integrierte** Sicht auf die Daten geboten werden muss. Etwaige Redundanzen werden also so weit wie möglich verhindert. Operative Datenbanken hingegen konzentrieren sich auf den eigenen Datenbestand, der **isoliert** von anderen Datenquellen betrachtet wird. Es ist zu erwähnen, dass operative Datenbanken meist als Quellen für analytische Datenbanken dienen.

Verfügen operative Datenbanken nur über **Detailldaten**, bieten analytische Datenbanken **detaillierte und aggregierte** Daten an. Dadurch wird es möglich, die entscheidungsunterstützende Analyse auf unterschiedlichen Abstraktionsstufen zu vollziehen. So kann man beispielsweise den Verkaufserlös vom einzelnen Verkauf ausgehend über das Tagestotal, Wochentotal, Monatstotal bis zum Jahrestotal hinauf verfolgen.

- ❖ **Verarbeitung:** Während bei operativen Datenbanken der Schwerpunkt auf **einfachen Schreib- und Lesetransaktionen** liegt, werden an analytische Datenbanken **komplexe Anfragen** gestellt. Der Schwerpunkt liegt somit auf einem lesenden Zugriff. Dennoch können in analytischen Datenbanken Ergänzungen (es werden die alten Werte beibehalten!) vorkommen. Diese sind aber weitaus weniger häufig als Änderungen in operativen Datenbanken. In operativen Datenbanken steht somit ein hoher Transaktionsdurchsatz im Vordergrund, für analytische Datenbanken ist die effiziente Bearbeitung komplexer Anfragen entscheidend.
- ❖ **Eigenschaften:** Auch hinsichtlich ihrer Eigenschaften setzen operative und analytische Datenbanken die Schwerpunkte anders. Ist für eine operative Datenbank entscheidend, dass jede Transaktion die Datenbank wieder in einen **konsistenten** Zustand überführt und dass **sämtliche** Transaktionen auch ausgeführt werden, steht bei analytischen Datenbanken die **Datenqualität** im Mittelpunkt.
- ❖ **Entwurf:** In operativen Datenbanken sind die konzeptionellen Einheiten die **Entitäten der Realwelt**. Um das Problem der Modifikationsanomalien zu umgehen, wird möglichst **redundanzarm** unter Berücksichtigung von **Normalformen** modelliert. Analytische Datenbanken hingegen orientieren sich bei der Modellierung an für die Entscheidungsfindung relevanten **Themenbereichen**; logisch zusammengehörende Informationen werden zusammen abgespeichert,

damit die Analyse möglichst optimal auf dem Datenbestand ablaufen kann. Um eine Effizienzsteigerung der Anfragebearbeitung zu erzielen, wird meistens auf eine **Normalisierung verzichtet**. Dies lässt sich aber aufgrund des vorwiegend lesenden Datenbankzugriffs riskieren.

- ❖ **Grösse:** Da analytische Datenbanken ihre Daten aus mehreren Quellen beziehen, können sie die Grösse operativer Datenbanken um ein Vielfaches übertreffen. Bewegen sich operative Datenbanken in den Bereichen von **Megabytes**, umfassen analytische Datenbanken zuweilen einige **Terabytes**.

Die aufgeführten Unterschiede zwischen operativen und analytischen Datenbanken legen es nahe, die beiden Datenbanken voneinander zu trennen. Nebst den vielen Vorteilen einer solchen Architektur wie beispielsweise der Entlastung operativer Datenbanken und einer gewissen Unabhängigkeit von der Verfügbarkeit der Datenquellen ist aber der hohe Aufwand für die Aufrechterhaltung zu erwähnen. Da die analytische Datenbank aufgrund der Trennung eine eigene Datenbasis halten muss, in die sämtliche Informationen aller Quellen eingebracht werden, fällt der Aufwand für den Aufbau, die Aktualisierung und die Wartung einer solchen replizierten Datenbasis schwer ins Gewicht.

2.1.2 Data Warehouse und Data Warehousing

Für **analytische Datenbanken** wird in dieser Arbeit auch der Begriff **Data Warehouse** synonym verwendet. Er wurde von Inmon in [Inmo1] folgenderweise geprägt: „*A data warehouse is a subject oriented, integrated, non-volatile, and time variant collection of data in support of management's decisions.*“ Diese Definition stimmt mit den in Tabelle 2-1 aufgeführten Punkten überein. Zusätzlich macht sie nochmals deutlich, dass aufgrund des vorwiegend lesenden Zugriffs und der Historisierung die Daten unverändert (non-volatile) bleiben.

Der Begriff **Data Warehousing** hingegen umfasst gemäss [Ditt] sämtliche „*Konzepte, Methoden, Werkzeuge und Systeme sowie deren Einsatz zur Nutzung von Daten für die Unterstützung von Entscheidungsfindungsprozessen.*“ Er geht somit über die eigentliche Datenbank bzw. das eigentliche Datenbankverwaltungssystem hinaus und bezieht auch die „Hintergrundarbeit“ zum Aufbau und zur Aufrechterhaltung eines Data Warehouse mit ein.

2.2 ARCHITEKTUR EINES DATA-WAREHOUSE-SYSTEMS

Abbildung 2-1 zeigt eine schematische Architektur für ein **Data-Warehouse-System**. Neben dem eigentlichen Data Warehouse, also der Datensammlung, umfasst ein solches System auch sämtliche Werkzeuge und Prozesse zur Akquisition und zum Zugriff auf die Daten. Ganz links im Bild sind die Datenquellen zu sehen, welche nebst operativen Datenbanken auch andere externe Quellen beinhalten können. Externe Daten werden eingekauft oder beschafft, um die eigenen Unternehmensdaten abzurunden. Der Datenfluss in Abbildung 2-1 geht grundsätzlich von links nach rechts: Die Daten werden aus den verschiedenen Quellen extrahiert und ins Data Warehouse geladen, von wo sie gelesen werden können. Unterschiedlichste Zugriffswerkzeuge erlauben dann die Interpretation der Daten und bereiten die daraus gewonnenen Informationen zu Entscheidungsunterstützung in geeigneter Weise auf. Unter Umständen werden durch die Manipulation der Daten auf Zugriffsebene neue Erkenntnisse gewonnen, die vorher nur implizit aus den Daten hervorgingen. Solche Erkenntnisse können dann nach Wunsch explizit ins Data Warehouse und eventuell in die ursprüngliche Quelle zurückgeschrieben werden. Da ein solcher Rückfluss aber nicht die Regel ist und quasi „gegen den Strom“ fliesst, ist er in Abbildung 2-1 nicht aufgeführt. In den folgenden Abschnitten werden nun die in der Abbildung grau schraffierten Felder behandelt.

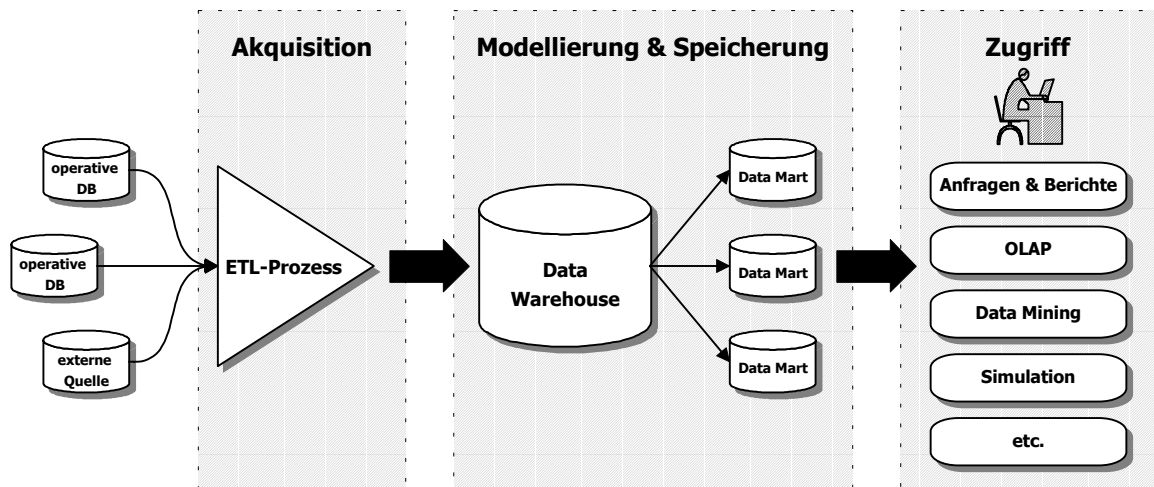


Abbildung 2-1: Architektur eines Data-Warehouse-Systems, Quelle: [Ditt]

2.2.1 Akquisition

Wie aus Abbildung 2-1 hervorgeht, erlaubt ein Data Warehouse den Zugriff auf Daten, die über mehrere Quellen zerstreut sind. Dadurch, dass ein Data Warehouse als gemeinsamer, grosser „Datenbehälter“ sämtliche Quelldaten umfasst und so einen einzigen zentralen Zugriffspunkt erzeugt, wird es erst möglich, die in mehreren Quellen versteckte Information für das Management zugänglich zu machen. Zudem werden die Datenquellen durch ein Data Warehouse entlastet, indem Anfragen für die Gewinnung von entscheidungsunterstützenden Informationen nicht an die eigentlichen Quellen sondern ans Data Warehouse gerichtet werden. Man kann sich somit in den Datenquellen allein auf die Optimierung des operativen Betriebs konzentrieren. Da die Datenquellen aufgrund ihrer Entstehungsgeschichte aber zumeist eine grosse Heterogenität aufweisen, können die Quelldaten nicht „eins zu eins“ ins Data Warehouse übernommen werden. Weiter muss unter Umständen auch eine Auswahl getroffen werden, da nicht alle Daten relevant sind. Dazu ist ein komplexer Akquisitionsprozess nötig, der die Daten aus den einzelnen Quellen zieht, sie in ein einheitliches Format bringt und sie schliesslich ins Data Warehouse einfügt. Dieser Prozess wird gemeinhin als **ETL-Prozess** bezeichnet, was für **Extraktion** (engl. **extraction**), **Transformation** (engl. **transformation**) und **Laden** (engl. **load**) steht. Diese drei Schlagworte bezeichnen gewissermassen die Hauptaktivitäten, die immer in einem Akquisitionsprozess anfallen. Es lassen sich aber noch weitere Aufgaben der Datenakquisition zuschreiben. Dazu gehören die mit der Transformation einhergehende **Integration** sowie die **Bereinigung**, **Vervollständigung** und **Historisierung**. Die nachfolgenden Abschnitte erläutern sämtliche Aufgaben der Datenakquisition. Die gewählte Reihenfolge der Präsentation ist dabei aber nicht zwingend, sondern kann je nach verfolgter Verfahrensweise unterschiedlich ausfallen.

Es ist noch festzuhalten, dass die Datenakquisition nicht ein einmaliger Prozess ist. Es wird zwischen dem initialen Laden eines Data Warehouse und seiner Aktualisierung unterschieden. Während man beim **initialen Laden** sämtliche Daten aus den Quellen extrahiert, transformiert und ins Data Warehouse lädt, versucht man aus Effizienzgründen, die **Aktualisierung** durch einen **inkrementellen Ansatz** zu bewältigen: Anstatt wiederum alle Daten ins Data Warehouse zu propagieren und dadurch die alten Daten zu überschreiben, werden nur die Änderungen der Datenquellen seit der letzten Aktualisierung berücksichtigt.

Extraktion

Die **Extraktion** hat zum Ziel, die für die Entscheidungsfindung relevanten Daten aus den Datenquellen zu filtern. Dabei muss der Heterogenität der Datenquellen und der Verarbeitung grosser Datenvolumina Rechnung getragen werden. Bei einer inkrementellen Aktualisierung des Data Warehouse müssen in diesem Schritt zusätzlich jene Daten ausfindig gemacht werden, die seit dem letzten Laden geändert haben. Die Extraktion kann zu unterschiedlichen Zeitpunkten bzw. in unterschiedlichen Abständen erfolgen. Während die **benutzergesteuerte** Extraktion erst auf explizites Verlangen hin durch den Benutzer durchgeführt wird, ist die **ereignisgesteuerte** und **periodische** Extraktion vom Eintreten eines Ereignisses bzw. dem Ablauf einer Zeitdauer abhängig. Die ereignisgesteuerte Extraktion setzt zudem eine Überwachung, ein so genanntes **Monitoring**, der Datenquellen voraus: Monitore überwachen die Datenquellen hinsichtlich Änderungen und initiieren dann den Extraktionsprozess. Sie können gemäss [Ditt] grundsätzlich danach unterschieden werden, ob sie lediglich auf stattgefundenen Änderungen in den Quellen hinweisen oder ob sie zusätzlich alle interessierenden Änderungen in einer Differenzdatei (sog. Delta-Datei) zur Verfügung stellen. Insofern können Monitore auch den nachfolgenden Extraktionsprozess vereinfachen, indem sie ihm das Ausfindigmachen der Änderungen abnehmen. Weitere Ausführungen zum Monitoring finden sich in [Ditt], [BaGü], [Kimb2], [Inmo1], [ChDa] und [Dev1].

Transformation und Integration

Aufgrund der Heterogenität der Quellen können die extrahierten Daten in den unterschiedlichsten Formaten vorliegen. Es ist nun Aufgabe dieses Akquisitionsschrittes, sämtliche Daten in ein einheitliches Format zu **transformieren**. Dazu werden die extrahierten Daten in einen **Arbeitsbereich** (engl. **staging area**) geladen, wo die notwendigen Transformationen dann vollzogen werden. Für die nachfolgenden Akquisitionsschritte verbleiben die Daten bis zum eigentlichen Laden in diesem Zwischenspeicher. Während dieser Transformation werden die Quelldaten auf die zur Verfügung stehenden Datenstrukturen des Data Warehouse abgebildet, wobei für die Abbildung auch Integrationsoperationen angewendet werden. Solche **Integrationsoperationen** haben zum Ziel, die lokalen Schemata der Quellsysteme zu einem gemeinsamen globalen Schema zu vereinen. Zudem werden durch die Integration Redundanzen vermieden und ein einheitlicher Detaillierungsgrad geschaffen.

Bereinigung

Wie bereits in Tabelle 2-1 festgehalten, spielt die Datenqualität in analytischen Datenbanken und somit in Data Warehouses eine zentrale Rolle: Der Erfolg einer Entscheidung steht und fällt mit der Qualität der zugrunde liegenden Daten. Es ist somit essentiell, etwaige qualitative Mängel in dieser Phase der **Datenbereinigung** auszumerzen. Inkompatible Datenformate und Inkonsistenzen in der Namensgebung sind nur wenige der zahlreichen Gründe, die zu Qualitätsproblemen führen können. Der für die Bereinigung anfallende Aufwand könnte natürlich beträchtlich reduziert werden, falls bereits auf Ebene der Quellsysteme qualitätssichernde Massnahmen ergriffen würden.

Vervollständigung

Nebst der Bereinigung müssen während der Akquisition die Daten auch **vervollständigt** werden. So werden beispielsweise abgeleitete und aggregierte Werte berechnet. Zudem muss die Handhabung für fehlende Werte und die damit einhergehende unterschiedliche Semantik definiert werden. Ein weiterer wichtiger Aspekt in der Datenvervollständigung ist die Vergabe von neuen Schlüsseln. Diese neuen, künstlich erzeugten **Schlüsselstellvertreter** (engl. **surrogate key**) sind im Data Warehouse global

eindeutig. Zu diesem Schritt ist man gezwungen, da Quellschlüssel nur im jeweiligen operativen System eindeutig sind und somit keine globale Eindeutigkeit haben. Zudem können sich unter Umständen die Quellschlüssel mit der Zeit ändern.

Historisierung

Wie bereits in Tabelle 2-1 erwähnt, ist die zeitbezogene Analyse der Datenbestände ein fundamentaler Aspekt im Data Warehousing. Hierzu benötigt man die **Historisierung** (siehe Kapitel 3). Sie erlaubt, die zeitliche Entwicklungsgeschichte der Daten festzuhalten, indem bei Änderungen nicht die alten Werte überschrieben, sondern die geänderten Werte als zusätzliche Einträge in die Datenbank aufgenommen werden. Die Historisierung der Daten passiert ebenfalls während der Datenakquisition, um sicherzustellen, dass nur historisierte Daten ins Data Warehouse gelangen. Es sei an dieser Stelle auch vermerkt, dass die Historisierung unter Umständen eine nochmalige Neuvergabe der Schlüssel erzwingt. Näheres dazu findet sich in Kapitel 3.

Laden

Nach erfolgreicher Beendigung sämtlicher vorhergehenden Akquisitionsaufgaben müssen schliesslich die transformierten, integrierten, bereinigten, vervollständigten und historisierten Daten ins Data Warehouse eingebracht, so genannt **geladen** werden. Hierzu wird das vom jeweiligen Datenbankverwaltungssystem zur Verfügung gestellte Ladewerkzeug verwendet, welches für die Verarbeitung grosser Datenmengen optimiert ist. Gleichzeitig können auch Indizes erstellt und die Daten sortiert werden. Der Ladevorgang kann prinzipiell offline oder online passieren. Während bei einem **Offline**-Ladevorgang der Data Warehouse Betrieb stillsteht, können im **Online**-Modus Anfragen ans Data Warehouse getätigt werden. Grundsätzlich geschieht das initiale Laden offline, inkrementelle Aktualisierungen hingegen online. Die Berücksichtigung historisierter Daten führt selbstverständlich dazu, dass während des Ladens Änderungen zusätzlich ins Data Warehouse eingebracht werden und nicht zum Überschreiben der alten Werte führen. Insofern bestimmt die Historisierung den Ladevorgang.

2.2.2 Modellierung und Speicherung

Wie Abbildung 2-1 zeigt, ist das Data Warehouse der Ort, an welchem die in der Akquisitionsphase gewonnenen Daten gespeichert werden, um einen zentralen Zugriff zu ermöglichen. Aufgrund des grossen Datenvolumens und der hohen Benutzeranzahl des Data Warehouse ist ein solcher singulärer Ansatz unter Umständen problematisch. Zudem würde die Realisation unterschiedlichster benutzergerechter Sichten die Verwendung mehrerer Speicher verlangen. Deshalb kommen oft zusätzlich zum eigentlichen Data Warehouse so genannte Data Marts zum Einsatz. Ein **Data Mart** ist ein weiterer Speicher, der eine Teilmenge der Data-Warehouse-Daten umfasst und eine spezifische benutzergerechte Sicht auf die Daten erlaubt. Aufgrund seiner Grösse und seiner Spezialisierung macht ein Data Mart einen schnelleren Zugriff möglich. So könnte man beispielsweise jede Unternehmensfunktion wie Marketing oder Controlling durch einen separaten Data Mart unterstützen. Andererseits wäre aber auch eine Orientierung nach Regionen denkbar, indem beispielsweise in einem international tätigen Unternehmen die Daten auf länderspezifische Data Marts aufgeteilt werden. Werden Data Marts zusätzlich eingesetzt, so finden sich im Data Warehouse meistens die Detaildaten, während in den Data Marts die daraus aggregierten und zusammengefassten Daten dominieren.

Bis anhin wurde der Begriff Data Warehouse als übergeordneter Begriff verwendet, welcher die Datensammlung ungeachtet der Tatsache bezeichnet, ob ein Data Warehouse durch Data Marts

ergänzt wird oder nicht. Sind keine weiteren Angaben aus dem Text zu entnehmen, soll er weiterhin in diesem Sinn verstanden werden.

Die Modellierung der Daten eines Data Warehouse bzw. eines Data Mart muss grundsätzlich der Heterogenität und dem Volumen der Daten gerecht werden sowie eine möglichst optimale Analyse zur Entscheidungsunterstützung erlauben. Da auf Zugriffsseite vor allem eine interaktive, multidimensionale Datenanalyse (siehe OLAP in Abschnitt 2.2.3) dominiert, kommt hier vor allem die **multidimensionale Datenmodellierung** zur Anwendung. Näheres dazu erfolgt im Abschnitt 2.3.

2.2.3 Zugriff

Aus Abbildung 2-1 geht hervor, dass die unterschiedlichsten Zugriffswerkzeuge und dadurch die unterschiedlichsten Benutzergruppen auf ein Data Warehouse zugreifen. Neben der Beantwortung von Ad-hoc-Anfragen, der Erstellung von Berichten und der Durchführung von Simulationen stehen auf Zugriffsebene vor allem OLAP und Data Mining im Vordergrund.

OLAP steht für **Online Analytical Processing** und bezeichnet eine interaktive, multidimensionale Datenanalyse. Aufgrund der Tatsache, dass OLAP zur Entscheidungsfindung eingesetzt wird, bildet es gewissermassen den Gegenpol zum Online Transaction Processing, wo die Unterstützung des operativen Geschäfts im Vordergrund steht. OLAP setzt eine multidimensionale Datenmodellierung (siehe Abschnitt 2.3) voraus.

Ziel des **Data Mining** hingegen ist die Gewinnung von Informationen, die nur implizit aus den zugrunde liegenden Daten hervorgehen. Zentrales Anliegen ist hier die Erkennung von Mustern in den Daten, um daraus Assoziationsregeln und Klassifikationen abzuleiten oder Cluster zu erkennen. Mit Hilfe des Data Mining lässt sich beispielsweise eine Warenkorbanalyse durchführen, die aufzeigt, welche Produkte ein Kunde meistens zusammen einkauft. So lässt sich beispielsweise die Assoziationsregel gewinnen, dass in 80 % der Fälle, in welchen ein Kunde Brot und Butter kauft, dieser auch noch Milch kaufen wird [Ditt].

2.3 MULTIDIMENSIONALE DATENMODELLIERUNG

Die multidimensionale Datenmodellierung ist im Data Warehousing dominierend. Es zeigte sich, dass in vielen Belangen eine multidimensionale Sichtweise auf die Daten vorteilhaft ist. Die folgenden Abschnitte geben zuerst eine konzeptuelle Einführung in das multidimensionale Datenmodell und gehen dann zur Speicherung eines solchen Datenmodells über.

2.3.1 Multidimensionales Datenmodell

Das **multidimensionale Datenmodell** erlaubt die Analyse von Kennzahlen wie beispielsweise dem Umsatz unter gleichzeitiger Betrachtung mehrerer Dimensionen. Abbildung 2-2 zeigt das Beispiel des Milchprodukte-Produzenten Moloko, der die Grossisten Denner, Waro, Coop und Volg beliefert und die Produktgruppen Joghurt, Butter, Milch, Käse und Eis herstellt. In Abbildung 2-2 wird der Umsatz in Zusammenhang mit den Dimensionen Zeit, Produkte und Grossisten gebracht. So kann man beispielsweise aus der Abbildung entnehmen, dass im Juli für CHF 53'000.- Eis an den Grossisten Denner geliefert wurde, während es im Februar lediglich CHF 11'000.- waren.

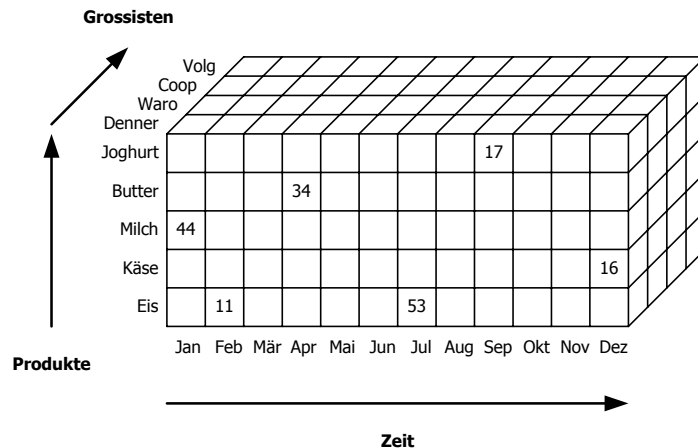


Abbildung 2-2: Multidimensionales Datenmodell

Die im Vordergrund der Analyse stehenden Kennzahlen wie der Umsatz im Beispiel werden in der multidimensionalen Datenanalyse als **Fakten** bezeichnet. Sie belegen in Abbildung 2-2 die Zellen des Quaders. Die **Dimensionen** hingegen erlauben unterschiedliche Sichtweisen auf diese Fakten: So lässt sich der Umsatz verschiedenen Produktgruppen zuordnen. Es lässt sich aber auch erkennen, welcher Umsatz mit welchem Grossisten erzielt wurde. Zudem kann man verfolgen, wie sich der Umsatz mit der Zeit verändert hat. Die Dimensionen spannen somit den Analyseraum auf, der dann die Verfolgung der Fakten entlang unterschiedlicher Achsen möglich macht und aufgrund seines Aussehens umgangssprachlich als Würfel bezeichnet wird. Meistens kommen für die Analyse mehr als drei Dimensionen in Frage, man spricht dann in diesem Fall von einem n-dimensionalen Würfel oder Hyperwürfel. Für die graphische Darstellung eines solchen Hyperwürfels muss dann immer eine Auswahl unter sämtlichen Dimensionen getroffen werden, da sich das menschliche räumliche Vorstellungsvermögen leider auf den dreidimensionalen Raum beschränkt. Während Fakten meist **numerische** und **additive** Werte sind, zeichnen sich Dimensionen hauptsächlich durch einen beschreibenden Charakter aus. Sie bilden gewissermassen die **Attribute** zu den Fakten und dienen deshalb als Einstiegspunkt bei Anfragen. Zudem macht auch deren Bezeichnung auf eine weitere Eigenschaft der Dimensionen aufmerksam: ihre **Orthogonalität**. Die einzelnen Dimensionen sind unabhängig voneinander und frei miteinander kombinierbar. Rein theoretisch kann jede Zelle in einem Hyperwürfel besetzt sein.

Nebst der Möglichkeit, Fakten entlang einer Dimension zu verfolgen, ist eine weitere wichtige Eigenschaft des multidimensionalen Modells die Fähigkeit, Fakten auf verschiedenen Abstraktionsstufen innerhalb einer Dimension zu analysieren. So lässt sich beispielsweise der Umsatz, wie in Abbildung 2-2 gezeigt, den einzelnen Produktgruppen zuordnen. Andererseits sollte es aber auch möglich sein, den totalen Umsatz über sämtliche Produktgruppen oder den Umsatz für jedes einzelne Produkt wie Schokoladejoghurt oder Camembert anzuzeigen. Dazu benötigt man aber eine hierarchische Ordnung der Dimensionselemente. Abbildung 2-3 zeigt eine solche **Hierarchie** für die Produktdimension. Jeder Knoten in diesem hierarchischen System bildet eine Abstraktionsstufe der Betrachtung. Das ganze System definiert so genannte **Konsolidierungspfade**, entlang deren man die Fakten **aggregieren** kann: Kennt man beispielsweise die Umsätze für jede einzelne Joghurtsorte wie Schokolade und Erdbeere, kann man daraus den Umsatz aller Joghurts berechnen. Dieser geht wiederum in den Umsatz aller Milchprodukte ein, welcher schliesslich die höchste Aggregationsstufe für die Fakten hinsichtlich der Dimension Produkte bildet. Die Ausführungen machen deutlich, warum es sich bei den Fakten vorzugsweise um numerische und additive Werte handeln sollte.

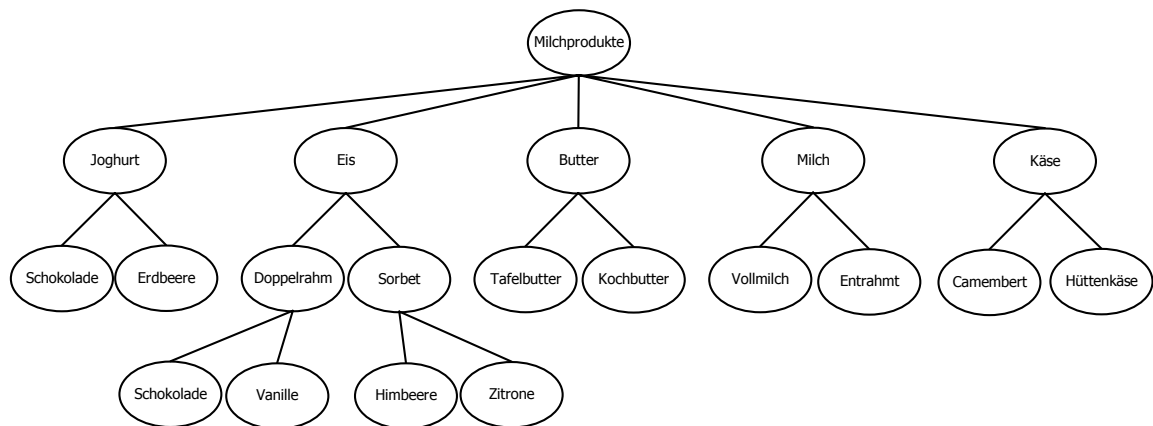


Abbildung 2-3: Hierarchische Ordnung der Dimensionselemente und Konsolidierungspfade

Die diskutierten Eigenschaften machen es nun dem Benutzer möglich, den Hyperwürfel in vielfältiger Weise zu manipulieren. Eine solche Interaktion wird typischerweise durch eine graphische Benutzeroberfläche unterstützt, die das Ergebnis einer Manipulation direkt am Bildschirm anzeigt. Man unterscheidet mehrere **multidimensionale Operationen**, die vom Drehen eines Hyperwürfels (Pivoting, Rotation) über die Aggregation entlang eines Konsolidierungspfad (Roll up & Drill down) bis zum „Herausschneiden“ einer Scheibe oder eines Teilwürfels (Slice & Dice) reichen [Ditt], [Kimb2] und [ChDa].

Abschliessend sei noch auf den **inhärenten Zeitbezug** des multidimensionalen Datenmodells verwiesen: Dadurch, dass die Zeit als weitere Dimension ins Modell eingeht, wird eine zeitbezogene Analyse der Fakten möglich.

2.3.2 Speicherung multidimensionaler Daten

Das in Abschnitt 2.3.1 vorgestellte multidimensionale Datenmodell kann nun in Abhängigkeit von der zugrunde liegenden Datenbank unterschiedlich umgesetzt werden. Man unterscheidet heutzutage prinzipiell zwischen einer relationalen und einer multidimensionalen Speicherung. Werden multidimensionale Daten durch ein relationales Datenbankverwaltungssystem verwaltet, verwendet man hierfür den Begriff **ROLAP, Relational Online Transaction Processing**. Bei Verwendung eines multidimensionalen Datenbankverwaltungssystem hingegen spricht man von **MOLAP, Multidimensional Online Transaction Processing**.

Aufgrund des Schwerpunkts dieser Arbeit auf relationale Datenbanken wird auf eine weitergehende Betrachtung von MOLAP verzichtet. Es sei an dieser Stelle auf [Ditt], [BaGü], [Kimb2], [ChDa] und [Dev1] verwiesen. Für eine relationale Speicherung multidimensionaler Daten kommen unterschiedliche Konzepte zur Anwendung. In dieser Arbeit wird jedoch nur eines davon diskutiert, nämlich das in der Praxis häufig zum Einsatz gelangende Star-Schema.

Star-Schema

Wie aus Abbildung 2-4 hervorgeht, werden im **Star-Schema** die Dimensionen und Fakten als separate Relationen modelliert. Die Verknüpfung der einzelnen Fakten mit den zugehörigen Dimensionselementen erfolgt hierbei über eine **Primär-Fremdschlüssel-Beziehung**, wobei die Fakten-Relation die Primärschlüssel der jeweiligen Dimensionselemente als Fremdschlüssel aufführen. Eine solche Verknüpfung muss dann zur Laufzeit mittels einer Join-Operation, einem so

genannten **Star-Join**, aufgelöst werden. Es ist zu beachten, dass nur die Fakten mit der Zeitdimension verknüpft sind, wodurch der Schwerpunkt der Historisierung auf den Fakten liegt (siehe Abschnitt 3.3.1).

Der Name des Star-Schemas wird aufgrund der sternförmigen Anordnung der Dimensions-Relationen um die Fakten-Relation in Abbildung 2-4 deutlich. Eine wichtige Eigenschaft des Star-Schemas ist hierbei, dass die einzelnen Dimensions-Relationen im Gegensatz zur herkömmlichen relationalen Modellierung **nicht normalisiert** sind. Dadurch können Anfragen an die Datenbank effizienter bearbeitet werden, weil man sich weitere Join-Operationen erspart. Mit der damit einhergehenden Redundanz kann man aufgrund des vorwiegend lesenden Zugriffs leben.

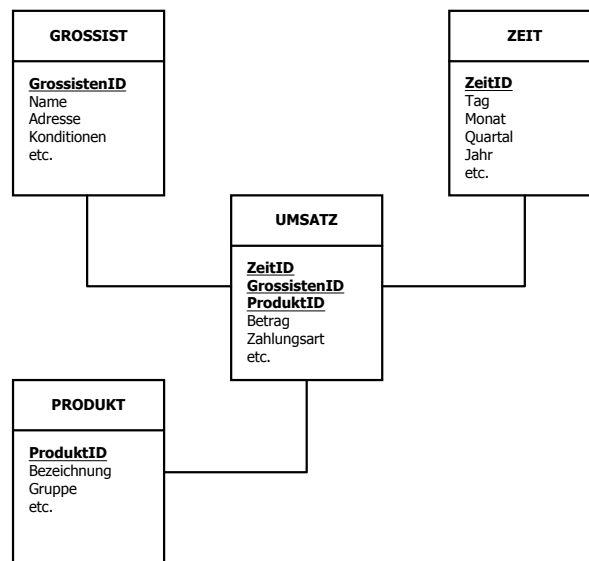


Abbildung 2-4: Star-Schema

2.4 METADATEN IM DATA WAREHOUSING

Die bereits in operativen Datenbanken bedeutende Rolle von **Metadaten** wird im Data Warehousing aufgrund seiner Komplexität existenziell. Während der Begriff in operativen Datenbanken vornehmlich für die Strukturbeschreibung der Datenbank, also für das Datenbankschema verwendet wird, ist er für analytische Datenbanken auszudehnen. Nebst der Beschreibung der eigentlichen Daten sind in Data Warehouses nämlich eine Unzahl weiterer Informationen zu verwalten. Darunter fallen beispielsweise die Angaben über die Datenquellen aber auch Erläuterungen zum Akquisitionsprozess. Es zeigt sich somit, dass Metadaten nebst einem **beschreibenden** auch einen **prozessbezogenen** Charakter aufweisen. Deshalb definieren [BaGü] Metadaten als „jede Art von Information, die für den Entwurf, die Konstruktion und die Benutzung eines Informationssystems benötigt wird“.

In [DiVa] wird die Forderung nach einer umfassenden **Verwaltung von Metadaten** geltend gemacht. Dazu sollen sämtliche anfallenden Metadaten in ein **Repository** eingebracht und durch einen **Metadaten-Manager** verwaltet werden. Abbildung 2-5 macht deutlich, dass der Zugriff auf ein solches Repository nur über den Metadaten-Manager geht: Die Einbringung etwaiger Metadaten aus den Datenquellen und die mögliche Auslagerung gewisser Metadaten ins Data Warehouse muss durch den Metadaten-Manager vollzogen werden. Zudem greifen der Warehouse-Manager selbst und diverseste Werkzeuge auf das Repository über den Metadaten-Manager zu. Hierbei stehen neben Werkzeugen für die Befriedigung von Benutzeranfragen auch Entwicklungswerkzeuge für den Entwurf neuer Anwendungen im Zentrum. Aber auch metadatengesteuerte Werkzeuge (siehe Kapitel 4), welche benötigte Steuerinformationen aus dem Repository beziehen, unterhalten eine

intensive Beziehung mit dem Metadaten-Manager. Dies verdeutlicht nochmals den beschreibenden als auch den prozessbezogenen Charakter von Metadaten und dadurch ihre zwei Nutzungsarten: Einerseits können sie **passiv** zur Systemdokumentation genutzt werden, andererseits werden sie **aktiv** als Steuerungsinformationen für metadatengesteuerte Werkzeuge eingesetzt.

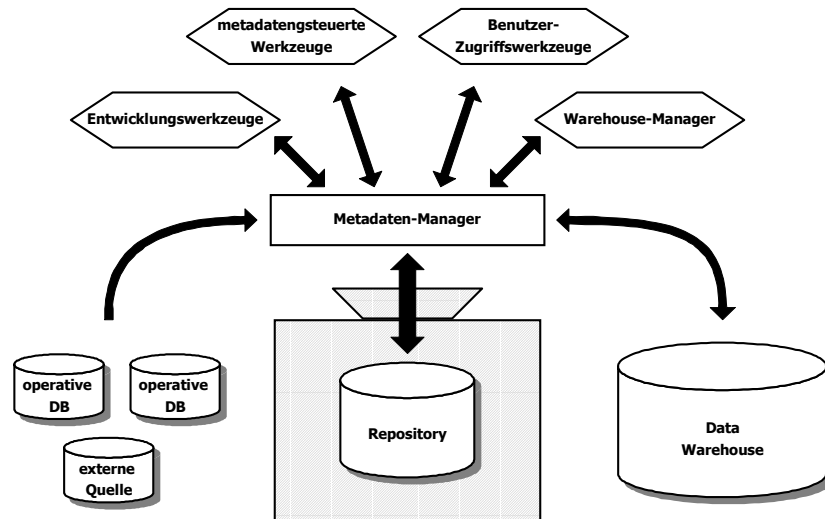


Abbildung 2-5: Metadatenverwaltung im Data Warehousing, Quelle: [Ditt]

Obwohl grundsätzlich eine **zentralisierte Metadatenverwaltung** anzustreben ist, bei welcher sämtliche Metadaten in einem einzigen Repository zur Verfügung stehen, ist ein solcher Ansatz in der Praxis oft nicht realisierbar. Das Vorhandensein mehrerer Repositories und die Heterogenität ihrer Entstehungsgeschichte zwingt einen leider nur allzu oft, einen anderen Weg einzuschlagen. Es verbleiben ein **dezentralisierter** und ein **föderierter** Ansatz. Während bei ersterem die einzelnen Repositories weitgehend autonom bleiben und lediglich den Metadatenaustausch mit Hilfe von Austauschstandards ermöglichen, erfolgt bei einer föderierten Metadatenverwaltung eine engere Bindung bzw. Abstimmung der Repositories untereinander, indem eine globale, konzeptuelle Sicht auf die Metadaten geboten wird. Eine föderierte Metadatenverwaltung ist deshalb als Mittelweg vorzuziehen.

3 HISTORISIERUNG

Ausgehend von in Literatur und Praxis vorherrschenden Ansätzen und Techniken versucht dieses Kapitel ein abstraktes Rahmenmodell für die Historisierung von Daten zu entwickeln.

Die **Historisierung** von Daten erlaubt, die **zeitliche Entwicklungsgeschichte** der Daten festzuhalten: Vergangene und unter Umständen auch zukünftige Zustände (siehe Abschnitt 3.1.2) sollen neben den aktuellen Einträgen aus dem Datenbestand hervorgehen. Grundsätzlich werden bei Änderungen alte Werte niemals überschrieben, sondern beibehalten. Die neuen Werte gehen in Form eines neuen, zusätzlichen Eintrags in die Datenbank ein. Somit ist es zu jeder Zeit möglich, die Historie einer Entität der Realwelt aufzurollen. Wird auf eine Historisierung verzichtet, widerspiegelt die Datenbank zu einem Zeitpunkt lediglich den aktuellen Zustand einer Entität, ihre Ausprägungen jenseits dieser Momentaufnahme gehen verloren.

Gerade im Umfeld von Data Warehouses, wo die zeitliche Betrachtung von Daten eine grundsätzliche Anforderung darstellt, erhält die Historisierung und deren Umsetzung eine fundamentale Bedeutung.

Leider findet man in der Literatur keine einheitliche Betrachtung der Historisierung. Vielmehr lassen sich einzelne Ansätze voneinander unterscheiden, die für ein spezifisches Anwendungsgebiet erarbeitet wurden. So kann man einerseits aus dem Gebiet der **temporalen Datenbanken** und andererseits aus demjenigen der **multidimensionalen Datenmodellierung** Historisierungsaspekte gewinnen. Es ist deshalb der hauptsächliche Beitrag der vorliegenden Arbeit, die verschiedenen praktizierten Historisierungstechniken unter einem einheitlichen Blickwinkel, dem **abstrakten Historisierungsmodell**, zu vereinen. Abschnitt 3.1 führt in die temporalen Datenbanken ein. Die in diesem Abschnitt erarbeiteten Historisierungskonzepte werden dann für die temporalen Erweiterungen von Datenbanken in Abschnitt 3.2 verwendet. Die multidimensionale Datenmodellierung und die damit verknüpften Fragen der Historisierung werden in Abschnitt 3.3 diskutiert. In Abschnitt 3.4 wird erörtert, ob auch Metadaten zu historisieren sind, und in Abschnitt 3.5 wird schliesslich das abstrakte Historisierungsmodell entwickelt.

3.1 HISTORISIERUNGSASPEKTE IN TEMPORALEN DATENBANKEN

In vielen Anwendungsbereichen von Datenbanken hat sich die Forderung, die zeitliche Entwicklungsgeschichte der Daten festhalten zu können, manifestiert: Versicherungsgesellschaften möchten beispielsweise über Änderungen der Versicherungssumme einer Police und über versicherte Ereignisse Buch führen. Im Gesundheitswesen will man die Krankengeschichte jedes einzelnen Patienten verfolgen können. Und in Finanzapplikationen sollte man schliesslich die zeitliche Veränderung des Kontostandes festhalten. Die Möglichkeit zur Historisierung sollte dabei vollumfänglich durch das Datenbankverwaltungssystem unterstützt werden. Forschung und Entwicklung haben hierauf mit dem Konzept der temporalen Datenbank geantwortet:

Eine **temporale Datenbank** (engl. **temporal database**) ist eine Datenbank, welche die **Historisierung von Daten** bewerkstelligt.

Von den temporalen Datenbanken lassen sich die so genannten **nicht-temporalen Datenbanken** (engl. **nontemporal database**, **snapshot database**) abgrenzen, welche nicht historisieren und somit lediglich den aktuellen Zustand repräsentieren. Obschon nicht-temporale Datenbanken keine Historie führen, finden sich auch dort gewisse temporale Aspekte wieder: Zur Beschreibung jener zeitlichen Eigenschaften, die integraler Bestandteil der Entität der Realwelt sind, stellt das System temporale Datentypen zur Verfügung. Beispiele für solche zeitliche Eigenschaften sind das Geburtsdatum einer Person oder die maximale Ausleihfrist in einer Bibliotheksverwaltung. All diesen Zeitangaben ist aber gemeinsam, dass sie nichts über die Entwicklungsgeschichte der Entität der Realwelt aussagen,

sondern lediglich eine weitere Beschreibung liefern. Sie stehen somit in keinem Bezug zur Historisierung. Zudem haben diese Angaben für das System keinerlei weitere Bedeutung und müssen durch den Benutzer interpretiert werden, weshalb man von **benutzerdefinierter Zeit** (engl. **user-defined time**) spricht [Ste].

In temporalen Datenbanksystemen werden zusätzlich zur benutzerdefinierten Zeit temporale Datentypen verwendet, welche das Datenbankverwaltungssystem für die Historisierung benötigt. Solche historisierungsspezifischen Datentypen werden durch das Datenbankverwaltungssystem interpretiert und unterscheiden sich dadurch klar von den benutzerdefinierten Zeitangaben. Neben dieser temporalen Erweiterung des Datenmodells muss ein temporales Datenbankverwaltungssystem zusätzlich auch über eine temporale Anfrage- und Manipulationssprache verfügen. Ein temporales Datenbankverwaltungssystem stellt somit geeignete Konstrukte zur Definition, Manipulation und Verwaltung der Datenentwicklungsgeschichte bereit. Eine bis anhin in den Anwendungen verborgene temporale Semantik wird nun im System explizit.

Die Betrachtung temporaler Datenbanken ist zur Beantwortung der in dieser Arbeit verfolgten Fragestellung in zweifacher Hinsicht hilfreich: Zum einen lassen sich Einblicke in etablierte Verfahren zur Historisierung von Daten gewinnen, zum anderen können temporale Datenbanken als Datenquellen im Data-Warehousing-Prozess auftreten.

Im Folgenden wird nun auf diejenigen Aspekte temporaler Datenbanken eingegangen, die für die Historisierung bedeutend sind.

3.1.1 Zeitrepräsentation

Obschon Zeit an sich kontinuierlich ist, muss man sie zur Verwendung in digitalen Rechnern durch diskrete Zeitpunkte darstellen. In temporalen Datenbanken wird die Zeit deshalb durch eine Folge geordneter Zeitpunkte modelliert. Die Dauer eines solchen Zeitpunktes, also die Granularität der Zeitrepräsentation, ist von der jeweiligen Anwendung abhängig: Die Historisierung von Adressen wird beispielsweise auf Stufe Tag passieren, während man bei Banktransaktionen zusätzlich auch die Uhrzeit erfassen will. In der Literatur wird für den Begriff Zeitpunkt auch der Begriff **Chronon** verwendet, welches das einer Anwendung zugrunde liegende Zeitgranulat bezeichnet.

Mit dieser Abstraktion weg von einer kontinuierlichen hin zu einer diskreten Zeitrepräsentation geht gezwungenermaßen ein gewisser Informationsverlust einher: Ereignisse, die innerhalb desselben Chronons liegen, erscheinen als simultan, obwohl sie dies in Wirklichkeit nicht sein müssen. Allerdings ist diese Problematik lediglich von theoretischem Interesse und hat für die Praxis keine Relevanz, da für die meisten Anwendungen die zur Verfügung stehenden Zeitgranulate genügend fein sind.

3.1.2 Gültigkeitszeit versus Transaktionszeit

Auf den ersten Blick erscheint es sinnvoll, die Historisierung von Daten so vorzunehmen, dass sich die Zeitangaben auf die Geschehnisse in der Realwelt beziehen. So soll beispielsweise ersichtlich sein, dass Frau Xiang seit dem 1. Januar 2000 an der Drachenstrasse 13 wohnt, zuvor jedoch während zehn Jahren am Lotosblütenweg 7 beheimatet war. Unter Umständen möchte man aber ebenfalls wissen, dass die Adressmutation erst am 5. Januar 2000 rückwirkend erfasst wurde.

Gültigkeitszeit. Beziehen sich in einer Datenbank die Zeitangaben auf den Zeitpunkt bzw. auf die Zeitdauer, in welchem die Datenwerte in der Realwelt Gültigkeit haben, so spricht man von **Gültigkeitszeit** (engl. **valid time**). Die Zeitangabe erfolgt normalerweise durch den Benutzer und kann durch ihn auch verändert werden. Im obigen Beispiel handelt es sich bei den Zeitangaben, von

wann bis wann Frau Xiang am Lotosblütenweg 7 bzw. an der Drachenstrasse 13 wohnte, um Gültigkeitszeit.

Transaktionszeit. Wird für die Historisierung hingegen jene Zeit(dauer) verwendet, zu welcher ein Wert in die Datenbank eingegangen ist bzw. in der Datenbank vorhanden war, benutzt man die so genannte **Transaktionszeit** (engl. **transaction time**). Die Transaktionszeit wird bei Erzeugung, Veränderung oder Entfernung eines Datenelementes durch das System generiert. Da sie die Zeit des „Datenbankeingriffs“ bezeichnet, ist sie sinnvollerweise durch den Benutzer nicht veränderbar. Für den Zeitpunkt der Adressmutation im Falle von Frau Xiang wurde die Transaktionszeit 5. Januar 2000 14:35:27 erfasst. Dieses Beispiel weist auch darauf hin, dass die Transaktionszeit üblicherweise in einer feineren Granularität als die Gültigkeitszeit angegeben wird.

Gültigkeits- und Transaktionszeit machen die Betrachtung der Datenentwicklungsgeschichte hinsichtlich unterschiedlicher Zeitachsen möglich: Die eine Zeitachse hält die Entwicklungsgeschichte der **Entität in der Realwelt** fest, die andere die Entwicklungsgeschichte der **Datenbank** selbst. Gültigkeitszeit und Transaktionszeit sind also **orthogonal** zueinander: sie sind getrennte Konzepte und können unabhängig voneinander verwendet werden (siehe Abbildung 3-1). Das folgende Zitat aus [Schä]¹ verdeutlicht diesen Aspekt sehr schön: „*The real world and the database move in the history along two completely independent, i.e. orthogonal, time axes, which do not even have to be measured in the same order of magnitude.*“ Aufgrund dieser Eigenschaft werden Gültigkeitszeit und Transaktionszeit in der Literatur oft als **Zeitdimensionen** bezeichnet. Wie auch im Kapitel 2 steht hier der Begriff Dimension für die Orthogonalität der beiden Zeitkonzepte. Gültigkeitszeit und Transaktionszeit können als separate Zeitdimensionen im multidimensionalen Datenmodell modelliert werden (siehe Abschnitt 3.3.1).

Beide Zeitdimensionen können entweder alleine oder gemeinsam eingesetzt werden. Werden sie gemeinsam verwendet, steigt die Ausdrucksfähigkeit beträchtlich: Zum einen werden **Korrekturen** als solche ersichtlich (siehe Abschnitt „Gültigkeits- und Transaktionszeit mit Zeitintervallen“, Seite 29), zum anderen geht hervor, ob die Wirksamkeit eines Datenbankeintrags **retroaktiv** (rückwirkend), **synchron** (gleichzeitig) oder **proaktiv** (zukünftig) zu seiner Transaktionszeit erfolgt. Dies wird in Abbildung 3-1² illustriert: Der Adresswechsel von Frau Xiang per 1. Januar 2000 wurde **retroaktiv** erst am 5. Januar 2000 erfasst. Der entsprechende Eintrag, symbolisiert durch den schwarzen Punkt, befindet sich in Abbildung 3-1 **unterhalb** der Winkelhalbierenden. Die Kündigung der alten Wohnung hingegen wurde **proaktiv** aufgenommen: Die Ungültigkeit der Adresse Lotosblütenweg 7 per 31. Dezember 1999 wurde bereits am 23. Dezember 1999 erfasst. Der entsprechende Eintrag befindet sich **oberhalb** der Winkelhalbierenden. Fallen Gültigkeits- und Transaktionszeit schliesslich zusammen, sind sie also zueinander **synchron**, liegt der entsprechende Punkt in Abbildung 3-1 auf der Winkelhalbierenden.

Aus Abbildung 3-1 geht weiter hervor, dass sich die Gültigkeitszeit im Gegensatz zur Transaktionszeit zusätzlich zur Gegenwart auch auf die Vergangenheit und Zukunft beziehen kann. Die Transaktionszeit hingegen bezeichnet die Zeit des Datenbankeingriffs und beschränkt sich aus diesem Grund auf die Gegenwart.

¹[Schä] verwendet die Begriffe **logische Zeit** (engl. **logical time**) und **physische Zeit** (engl. **physical time**) für Gültigkeits- und Transaktionszeit.

² In Abbildung 3-1 wurde der Übersichtlichkeit wegen auf die Zeitangabe bei der Transaktionszeit und auf die Verankerung des Ursprungs verzichtet.

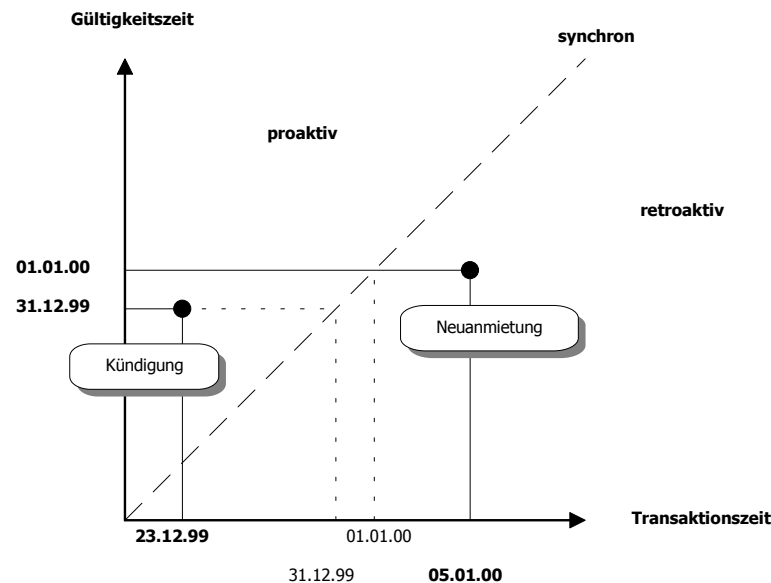


Abbildung 3-1: Orthogonalität von Gültigkeits- und Transaktionszeit

Typen temporaler Datenbanken. Temporale Datenbanken lassen sich danach typisieren, welche Zeitdimension für die Historisierung verwendet wird. Wird lediglich mit der **Gültigkeitszeit** gearbeitet, spricht man von **historischen Datenbanken** (engl. **historical database**) [Steil] oder von **Gültigkeitszeit-Datenbanken** (engl. **valid time database**) [ElNa]. In diesem Fall wird in Abbildung 3-1 entlang der Ordinate historisiert. Genügt die **Transaktionszeit** allein, stösst man in der englischen Literatur auf den Begriff **Rollback Database**. Dieser Begriff soll die Fähigkeit verdeutlichen, dass jeder vergangene Datenbankzustand wieder aufgerollt werden kann. [ElNa] verwenden in diesem Kontext zusätzlich die Bezeichnung **Transaktionszeit-Datenbank** (engl. **transaction time database**). In einer solchen Datenbank erfolgt die Historisierung in Abbildung 3-1 entlang der Abszisse. Werden schliesslich **beide Zeitdimensionen** gemeinsam eingesetzt, führt dies zu einer **bitemporalen Datenbank** (engl. **bitemporal database**). Nun kann jeder Punkt in dem durch die zwei Zeitdimensionen aufgespannten Raum in der Datenbank abgebildet werden.

3.1.3 Momentaufnahme versus Zustandsdauer

Um den Entwicklungsprozess einer Entität oder der Datenbank selbst wiederzugeben, werden in temporalen Datenbanken Momentaufnahmen oder Angaben über die Zustandsdauer verwendet. Hierbei ist die Wahl des Modellierungskonzeptes – Momentaufnahme versus Zustandsdauer – von der Art des Entwicklungsprozesses abhängig.

Entwicklungsprozess einer Entität in der Realwelt. Der Entwicklungsprozess einer Entität in der Realwelt kann in Anlehnung an [Schä] folgendermassen typisiert werden:

- ❖ **Konstant:** Die Entität verbleibt während ihrer ganzen Lebenszeit in demselben Zustand, es findet kein Entwicklungsprozess im eigentlichen Sinn statt. Ein Beispiel hierfür könnte das Geschlecht einer Person sein, vorausgesetzt man sieht von Geschlechtsumwandlungen ab.
- ❖ **Stufenweise konstant:** Ähnlich dem konstanten Entwicklungsprozess verbleibt die Entität während einer bestimmten Zeitdauer in einem konstanten Zustand. Im Unterschied zu Ersterem finden aber Zustandswechsel statt, die durch den Eintritt eines Ereignisses ausgelöst werden. Der Verlauf der Werte über die Zeit lässt sich sehr schön als Treppenfunktion abbilden. Beispiele für

solche Entwicklungsprozesse finden sich bei personenbezogenen Angaben wie Zivilstand oder Adresse.

- ❖ **Kontinuierlich:** Im Gegensatz zu einer stufenweise konstanten Entwicklung, bei welcher sich ein Zustand immer über eine Zeitdauer erstreckt, beziehen sich bei einem kontinuierlichen Entwicklungsprozess die Zustandsangaben lediglich auf einen Zeitpunkt. Bei einem kontinuierlichen Entwicklungsprozess ändert sich der Zustand einer Entität zudem stetig und nicht nur, wie dies bei stufenweise konstanten Entwicklungsprozessen der Fall ist, zu bestimmten Zeitpunkten. Der Werteverlauf wird durch eine kontinuierliche Kurve beschrieben. Ein Beispiel für diese Kategorie ist die Temperaturentwicklung in einem wissenschaftlichen Experiment.

Abgesehen vom konstanten Entwicklungsprozess erfordern die restlichen zwei Fälle ihre Historisierung und können durch Momentaufnahmen oder durch die Angabe der Zustandsdauer modelliert werden. Die Zeitangaben erfolgen in Gültigkeitszeit.

Momentaufnahme. Beschränkt man sich darauf, den Entwicklungsprozess lediglich **punktuell** wiederzugeben, beschreiben die Angaben den Zustand der Entität im Moment der Aufnahme. Die Gültigkeit der Angaben sind auf den Aufnahmezeitpunkt begrenzt, über den Zustand „links“ und „rechts“ davon wird keine Aussage gemacht. Jedem Eintrag in der Datenbank wird der **Zeitpunkt** der Momentaufnahme zugeordnet.

Zustandsdauer. Andererseits lässt sich auch angeben, **wie lange** eine Entität in einem bestimmten Zustand verbleibt. Für eine solche Betrachtungsweise werden den Einträgen in der Datenbank typischerweise Intervalle oder temporale Elemente zugefügt:

- ❖ **Intervalle** werden durch Angabe eines **Start- und Endzeitpunktes** gebildet. Hierbei können entweder beide Grenzpunkte explizit angegeben werden oder es genügt, nur einen der beiden aufzuführen, da sich der andere aus dem Kontext bestimmen lässt. Man spricht dann von einer impliziten Intervallangabe. Je nach Interpretation gehören die Intervallgrenzen zum Intervall dazu oder nicht dazu. Innerhalb eines Intervalls verbleibt die Entität im angegebenen Zustand, der Zustandswechsel erfolgt an den Intervallgrenzen. Somit muss mit jeder festzuhaltenden Zustandsänderung ein neuer Eintrag mit entsprechenden Intervallangaben erzeugt und in den Datenbestand eingeführt werden. Da Intervalle definitionsgemäss fortlaufend sind, kommt es auch dann zu einem neuen Eintrag, wenn der Zustand zwar derselbe ist, dieser aber während zweier, sich nicht berührender Zeitintervalle angenommen wird. In einem solchen Fall führt die Verwendung von Intervallen zu **Redundanz**.
- ❖ **Temporale Elemente** verhindern die im vorangehenden Punkt angesprochene Redundanz dadurch, dass sie die Angabe einer Menge von Intervallen erlauben. Ein und demselben Dateneintrag können also mehrere, sich nichtberührende Intervalle, also ein temporales Element, zugeordnet werden.

Kontinuierliche Entwicklungsprozesse lassen sich durch **Momentaufnahmen** modellieren. Die Aufnahmefrequenz bestimmt dann die Wiedergabetreue der Abbildung. Für stufenweise konstante Entwicklungsprozesse bieten sich sowohl Momentaufnahmen als auch Intervalle bzw. temporale Elemente an. Die Auswahl hängt davon ab, wie eine Entität auf eine Datenbank abgebildet wird, d.h. ob eine zustandsorientierte Modellierung oder eine ereignisorientierte Modellierung gewählt wird.

Da bei **stufenweise konstanten Entwicklungsprozessen** der Zustandswechsel aufgrund eines eintretenden Ereignisses erfolgt, hat man zwei Modellierungsmöglichkeiten: Entweder man hält den Zustand direkt in der Datenbank fest oder man speichert lediglich die ereignisauslösende Transaktion. Erstere Modellierungsmöglichkeit ist **zustandsorientiert**, letztere **ereignisorientiert**. Das Beispiel eines Bankkontos illustriert dies sehr schön: Bei einer zustandsorientierten Modellierung wird nach jeder Transaktion der aktuelle Kontostand ermittelt und in die Datenbank geschrieben. Bei einer ereignisorientierten Modellierung hingegen werden lediglich die einzelnen Gutschriften und Belastungen festgehalten und somit auf ein explizites Führen des Kontostandes verzichtet. Da der Kontostand aber jederzeit durch Summieren der einzelnen Transaktionen berechnet werden kann, ist

die Zustandsinformation implizit dennoch vorhanden. Im Falle eines Bankkontos ist natürlich die ereignisorientierte Modellierung vorzuziehen, da sie auf eine fortwährende und teure Berechnung des Kontostandes nach jeder einzelnen Transaktion verzichtet. Zudem würde ein explizites Führen des Kontostandes bei einem Fehler in unzähligen Folgefehlern enden. Die Anwendung bestimmt somit massgeblich die zu wählende Modellierung.

Für die Abbildung eines **zustandsorientierten, stufenweise konstanten Entwicklungsprozesses** sind **Intervalle** und **temporale Elemente** ideal, da sie die Angabe der Zustandsdauer erlauben. Bei einer **ereignisorientierten Modellierung von stufenweise konstanten Entwicklungsprozessen** hingegen sind **Momentaufnahmen** vorzuziehen, da sich die Angaben in der Datenbank lediglich auf den Moment des Ereigniseintritts beziehen. Betrachtet man den zeitlichen Werteverlauf der ereignisauslösenden Transaktion, so lässt sich dieser durch mehrere, nicht miteinander verbundene Punkte darstellen. Aus diesem Grund spricht [Schä] von einem diskreten Entwicklungsprozess. Im Gegensatz zu [Schä] wird aber in dieser Arbeit die diskrete Entwicklung nicht als vierter Entwicklungstyp aufgeführt, sondern als Folge einer ereignisorientierten Modellierung von einer stufenweise konstanten Entwicklung verstanden. Dieser Standpunkt beruht auf der Tatsache, dass einem so genannten diskreten Entwicklungsprozess eine andere Betrachtungseinheit zu Grunde liegt: Bei den weiter oben aufgeführten Entwicklungstypen wird die Entität selbst ins Auge gefasst, bei einer diskreten Entwicklung wird die ereignisauslösende Transaktion betrachtet.

Entwicklungsprozess der Datenbank. Der Entwicklungsprozess der Datenbank selbst zeichnet sich als **stufenweise konstant** aus: Mit jeder Modifikationsoperation (Einfügen, Ändern, Löschen) geht die Datenbank in einen neuen Zustand über. Da eine Datenbank aber **zustandsorientiert** ist, bieten sich zur Entwicklungsmodellierung lediglich **Intervalle** oder **temporale Elemente** an. Diese Zeitdauer wird in Transaktionszeit erfasst.

Abschliessend sei vollständigkeitshalber auf weitere Aspekte der Verwendung von Zeitintervallen verwiesen, die in der Literatur diskutiert werden. So erwähnt [Ste] den Einsatz von Mengenoperationen auf Intervalle und [Inmo3] zeigt auf, dass sich Intervalle hinsichtlich ihrer Aneinanderreihung in berührende (continuous), nichtberührende (non-continuous) und gegenseitig überlappende (overlapping) klassifizieren lassen. Zudem weist [Inmo3] darauf hin, dass es im Verlauf der Zeit aufgrund von Modifikationsoperationen zu einer unnötigen Zerlegung eines Intervalls in Teilintervalle kommen kann, was er als Data Fracturing bezeichnet.

3.1.4 Tupel-Versionierung versus Attribut-Versionierung

Es stellt sich nun die Frage, welches die **Granularitätsstufe** der Historisierung ist. Die Granularitätsstufe bestimmt jene Einheit, welche bei jeder Änderung völlig neu erzeugt und zusätzlich in die Datenbank eingefügt werden muss. In der Literatur über temporale Datenbanken werden vor allem die Stufen Attribut und Tupel diskutiert.

Tupel-Versionierung (engl. tuple versioning). Erfolgt die Historisierung auf Tupelebene, wird mit jeder Änderung ein vollkommen neues, aufdatiertes Tupel generiert, welches zusätzlich in die Datenbank eingetragen wird. Der Ausdruck Tupel soll in diesem Zusammenhang als generischer Begriff verstanden werden. Er bezeichnet gewissermassen die Dateneinheit. Je nach Kontext kann er sinngemäss interpretiert werden: Im Relationenmodell steht er für ein Tupel, im objektorientierten Umfeld hingegen für das Objekt selbst.

Attribut-Versionierung (engl. attribute versioning). Bei der Attribut-Versionierung hingegen wird lediglich das durch die Änderung betroffene Attribut aufdatiert, dessen neuer Wert dann als zusätzlicher Eintrag in den Datenbestand eingeht.

Grundsätzlich sollten beide Historisierungsverfahren wohlüberlegt eingesetzt werden und sich an der Quelle der Änderung orientieren: Gehen Änderungen hauptsächlich von einzelnen Attributwerten aus, ist eine Attribut-Versionierung zu favorisieren. In einem solchen Fall kann man sich nämlich das

redundante Führen der gleichbleibenden Attributwerte, wozu man bei der Tupel-Versionierung gezwungen wäre, ersparen. Zudem geht so die Geschichte jedes einzelnen Attributwertes klar hervor. Bezieht sich hingegen eine Änderung auf das gesamte Tupel, ist der Tupel-Versionierung den Vorzug zu geben. Dies ist beispielsweise dann der Fall, wenn man einzelne Produktversionen historisieren will. Gerade in einem solchen Beispiel kann es durchaus sinnvoll sein, dass man beide Historisierungsverfahren miteinander kombiniert: Marginale Veränderungen innerhalb einer Produktversion werden auf Attributebene historisiert, für die Historie der Produktversionen selber wird die Tupel-Versionierung eingesetzt. Natürlich gibt dann die Tupel-Versionierung den äusseren Zeitrahmen für die Attribut-Versionierung vor: die Zeitangabe für die Attributwerte dürfen die „Lebenszeit“ der einzelnen Produktversion nicht überschreiten.

Da die Granularitätsstufe als Historisierungseinheit üblicherweise auch jene Einheit bestimmt, auf welche sich die Zeitangaben beziehen, spricht man auch von **Tupel-Zeitstempelung** (engl. **tuple timestamping**) und **Attribut-Zeitstempelung** (engl. **attribute timestamping**). Sinnbildlich sind dies jene Einheiten, denen ein Zeitstempel „aufgedrückt“ wird. Weil dies aber nicht immer der Fall ist (siehe Abschnitt 3.2.1, Seite 21), werden in dieser Arbeit ausschliesslich die Begriffe Tupel- und Attribut-Versionierung verwendet.

3.2 TEMPORALE ERWEITERUNGEN

Nun ist es an der Zeit, die in den vorangehenden Abschnitten besprochenen Historisierungskonzepte auf Datenbanken anzuwenden. Abschnitt 3.2.1 befasst sich mit der Anwendung auf relationale Datenbanken und Abschnitt 3.2.2 mit derjenigen auf Datenbanken, die komplexe Strukturen unterstützen.

3.2.1 Temporale Erweiterung einer relationalen Datenbank

Vorliegender Abschnitt zeigt anhand von Beispielen auf, wie eine nicht-temporale, relationale Datenbank strukturell um temporale Aspekte erweitert werden kann, damit die Historie einer Entität ersichtlich wird. Dazu werden die Historisierungskonzepte aus Abschnitt 3.1 verwendet. Grundsätzlich sind sämtliche Konzepte auf relationale Datenbanken anwendbar, für die Attribut-Versionierung ist aber folgender Einwand geltend zu machen:

Da sich ein Relationenschema definitionsgemäss in **erster Normalform** befinden muss, was also das Vorkommen von zusammengesetzten und mehrwertigen Attributen untersagt, ist die Attribut-Versionierung nur „über einen Umweg“ realisierbar: Analog zum üblichen Vorgehen bei mehrwertigen Attributen ist für jedes zu historisierende Attribut eine zusätzliche Relation zu schaffen. In Anbetracht der grossen Anzahl an potentiell zu historisierenden Attributen und der daraus resultierenden Unmenge an Joins ist eine solche Implementation aber teuer erkauft. Aus diesem Grund sollte man bei relationalen Datenbanken eine Attribut-Versionierung nur dann einsetzen, wenn sich die Anzahl der zu historisierenden Attribute in Grenzen halten.

Diese **faktische Einschränkung auf die Tupel-Versionierung** hat dann keine weiteren Folgen, wenn die Historisierung des ganzen Tupels im Vordergrund steht. Gehen die Änderungen aber von den Attributwerten aus, was in den Anwendungen weitaus häufiger der Fall ist, hat die Verwendung der Tupel-Versionierung die folgenden Nachteile (siehe auch Abschnitt „Gültigkeitszeit mit Zeitintervallen“, Seite 24):

- ❖ Da bei der Tupel-Versionierung mit jeder Änderung auch nur eines Attributwertes ein neues Tupel generiert werden muss, kommt es zu **Redundanz**: Die alten, weiterhin gültigen Werte werden im neuen Tupel erneut geführt. Diese Redundanz wird dann umso schwerwiegender, wenn

Attributwerte häufig und unabhängig voneinander mutieren. Nebst dem Nachteil des erhöhten Speicherplatzbedarfs birgt Redundanz generell die Gefahr von Modifikationsanomalien in sich.

- ❖ Die **Historie der einzelnen Attributwerte ist nicht mehr explizit**: Da sich die Zeitangabe auf das gesamte Tupel bezieht, muss die Information, zu welchem Zeitpunkt bzw. während welcher Zeitdauer ein Attributwert gültig war, durch Vergleiche der Tupel untereinander abgeleitet werden. Möchte man diesen Informationsverlust wettmachen, ist für jedes zu historisierende Attribut separat eine Zeitangabe zu setzen. Bei diesem Vorgehen fällt jedoch die Einheit, auf welche sich die Zeitangabe bezieht, nicht mehr mit der Historisierungseinheit zusammen: Obwohl die Zeitangaben für die einzelnen Attributwerte gelten, wird mit jeder Mutation eines einzelnen Attributwertes ein neues Tupel erzeugt.
- ❖ Aufgrund des Vorgehens, dass mit jeder Änderung auch nur eines Attributwertes ein neues Tupel erzeugt wird, können Informationen über ein und dieselbe Entität der Realwelt über mehrere Tupel zerstreut werden. Da in relationalen Datenbanken allein das Tupel die grundlegende Dateneinheit zur Widerspiegelung einer Entität ist, kann die konzeptionelle Zusammengehörigkeit mehrerer Tupel nicht ausgedrückt werden. [Ste] bezeichnet diesen Semantikverlust als **vertikale, temporale Anomalie** (engl. **vertical temporal anomaly**).

Die folgenden Abschnitte zeigen die Anwendung der vorgestellten Historisierungskonzepte auf eine relationale Datenbank anhand von Beispielen. Sie orientieren sich dabei grundsätzlich an den in [ElNa] ausgearbeiteten Verfahren. Aufgrund der erwähnten Nachteile wird auf die Attribut-Versionierung verzichtet. Ebenso finden temporale Elemente keine Verwendung, da sie zu mehrwertigen Attributen führen. Es würden folglich die restlichen Kombinationsmöglichkeiten verbleiben. Da in dieser Arbeit aber die Ansicht vertreten wird, dass die Transaktionszeit lediglich zur Modellierung der Entwicklungsgeschichte der Datenbank verwendet werden sollte und es grundsätzlich immer einer Entwicklungsmodellierung der Entität bedarf, wird die Transaktionszeit in den Beispielen nur in Kombination mit der Gültigkeitszeit verwendet. Aus dieser Sichtweise handelt es sich bei den in der Literatur diskutierten Rollback Databases (siehe Abschnitt 3.1.2) eigentlich um historische Datenbanken, da sie im Prinzip den Entwicklungsprozess der Entität wiedergeben. Der Begriff Rollback Database und damit die vermeintliche Verwendung von Transaktionszeit lassen sich insofern rechtfertigen, dass die Zeitangaben bei einer solchen Datenbank wahrscheinlich vom System selbst generiert werden und in einer feineren Granularität erfolgen.

Insbesondere illustrieren die nachfolgenden Beispiele die Erweiterung des relationalen Datenbankschemas um historisierungsspezifische Angaben und das Vorgehen bei Modifikationsoperationen (Einfügen, Ändern, Löschen). Die diskutierten Erweiterungen sind somit lediglich struktureller Natur, eine etwaige temporale Erweiterung der Datenanfrage- und Manipulationssprache wird nicht berücksichtigt. Insofern handelt es sich in den Beispielen um keine echten temporalen Datenbanken (siehe Abschnitt 3.1). Die zugrunde liegende Beispielanwendung modelliert das Verhältnis einer Bank zu ihren Kunden und kann durch das in Abbildung 3-2 ersichtliche ER-Diagramm beschrieben werden. Dieses berücksichtigt noch keine temporalen Aspekte. Das dazugehörige relationale, nicht-temporale Schema ist in Tabelle 3-1 ersichtlich. Natürlich konzentriert sich die Beispielanwendung auf die Modellierung des Wesentlichsten, um daran die interessierenden Aspekte illustrieren zu können. In der Praxis würde das ER-Diagramm und das zugehörige Relationenschema natürlich viel umfangreicher ausfallen.

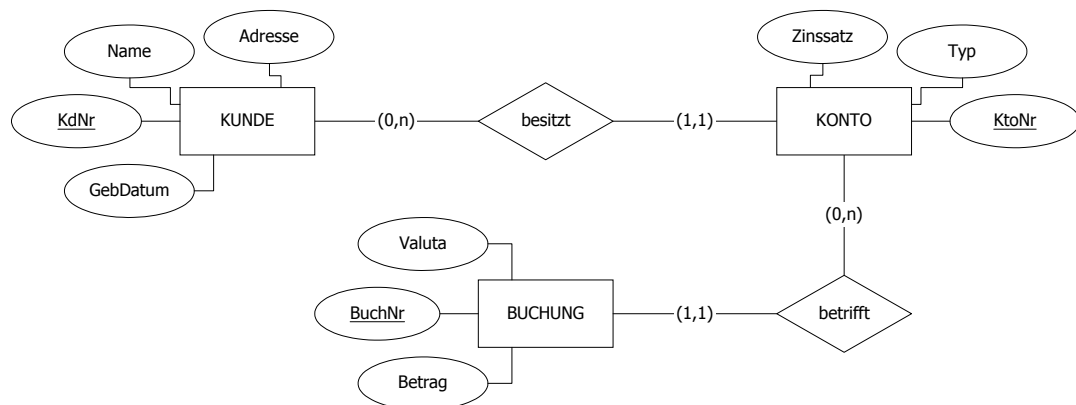


Abbildung 3-2: Nicht-temporales ER-Diagramm für die Beispielanwendung

KUNDE

<u>KdNr</u>	NName	VName	Strasse	PLZ	GebDatum
-------------	-------	-------	---------	-----	----------

WOHNORT

<u>PLZ</u>	Wohnort
------------	---------

KONTO

<u>KtoNr</u>	Typ	KdNr
--------------	-----	------

KONTOTYP

<u>Typ</u>	Bezeichnung	Zinssatz
------------	-------------	----------

BUCHUNG

<u>BuchNr</u>	Betrag	Valuta	KtoNr
---------------	--------	--------	-------

Tabelle 3-1: Nicht-temporales Relationenschema für die Beispielanwendung³**Gültigkeitszeit mit Zeitintervallen**

Tabelle 3-2 zeigt die temporale Erweiterung des Relationenschemas KUNDE. Da die Entwicklung einer Entität Kunde stufenweise konstant ist und zustandsorientiert modelliert wurde, werden zur Historisierung Zeitintervalle verwendet. Dazu weist die Relation KUNDE die zwei neuen Attribute **GültigAb** und **GültigBis** auf, welche die Intervallgrenzen bilden. Innerhalb dieser Grenzen entsprechen die zugehörigen Angaben der Realwelt, weshalb die Zeitangaben in Gültigkeitszeit erfolgen und durch den Benutzer geliefert werden müssen. Der spezielle Wert now für GültigBis wird immer dann verwendet, wenn der Endzeitpunkt noch nicht bekannt ist. Typischerweise kommt in einem Datenbanksystem dafür der maximal mögliche Datumswert zum Einsatz (z.B. 31.12.4712). Grundsätzlich lassen sich aktuelle Tupel dadurch ermitteln, dass man diejenigen Tupel herausfiltert, deren Intervallangaben den Betrachtungszeitpunkt enthalten. Da aufgrund der Historisierung der ursprüngliche Primärschlüssel KdNr aus Tabelle 3-1 nicht mehr eindeutig ist, bilden fortan die Attribute KdNr und GültigAb zusammen den **neuen Primärschlüssel**. Falls aber der ursprüngliche nicht-temporale Primärschlüssel KdNr mit der Zeit ändern kann, sollte man einen Schlüsselstellvertreter (siehe Abschnitt 2.2.1, Seite 7) wählen, der dann zusammen mit GültigAb zum

³ Die Bezeichnung „nicht-temporal“ ist insofern nicht ganz korrekt, da die Relation BUCHUNG bereits um Historisierungskonzepte erweitert ist (siehe Abschnitt „Gültigkeitszeit mit Momentaufnahmen“, Seite 25).

neuen Primärschlüssel kombiniert wird. Natürlich könnte anstatt GültigAb auch GültigBis für den neuen Primärschlüssel gewählt werden. Beim Attribut GebDatum handelt es sich um eine benutzerdefinierte Zeitangabe (siehe Abschnitt 3.1), welche hinsichtlich der Historisierung nicht weiter zu betrachten ist.

KUNDE

<u>KdNr</u>	NName	VName	Strasse	PLZ	GebDatum	<u>GültigAb</u>	GültigBis
1	Xiang	Lian	Lotosblütenweg 7	8002	13.11.41	01.01.90	31.12.99
1	Xiang	Lian	Drachenstrasse 13	8004	13.11.41	01.01.00	now
2	Eiffel	Pierre	Am Triumphbogen	8200	14.07.63	01.10.00	now
3	Tell	Köbi	Käshaldenstrasse 3	8052	01.08.35	01.01.60	30.09.74
3	Tell	Köbi	Läckerligasse 1	4051	01.08.35	01.10.74	31.12.86
3	Tell	Köbi	Käshaldenstrasse 3	8052	01.08.35	01.01.87	now
4	Kurz	Hilde	Hopfenweg 34	3011	22.06.52	01.05.91	30.09.94

WOHNORT

<u>PLZ</u>	Wohnort
3011	Bern
4051	Basel
8002	Zürich
8004	Zürich
8052	Zürich
8200	Schaffhausen

Tabelle 3-2: Gültigkeitszeit mit Zeitintervallen

Im Folgenden wird nun gezeigt, welchen Einfluss die Verwendung von Zeitintervallen in Gültigkeitszeit auf die Modifikationsoperationen haben:

Einfügen. Wird ein neuer Eintrag in die Datenbank eingefügt, müssen nun zusätzlich die Werte für die Attribute GültigAb und GültigBis gesetzt werden. Beide Angaben müssen vom Benutzer geliefert werden, da nur er alleine weiss, während welcher Zeitdauer der zu modellierende Sachverhalt in der Realwelt gilt. Da zum Einfügezeitpunkt der Endzeitpunkt meistens noch nicht bekannt ist, trägt GültigBis üblicherweise den Wert now. Ein Beispiel für das Einfügen eines neuen Tupels ist Pierre Eiffel. Bis jetzt gibt es nur ein einziges Tupel zu seiner Person.

Ändern. Die Einträge über Lian Xiang und Köbi Tell sind Beispiele für Änderungsoperationen. Als Frau Xiang am 1. Januar 2000 vom Lotosblütenweg 7 an die Drachenstrasse 13 zog, wurde in ihrem damaligen Eintrag GültigBis von now auf 31.12.99 geändert und ein neues Tupel mit den neuen Angaben eingefügt. Dieses neue Tupel hat für GültigAb den Wert 01.01.00 und für GültigBis den Wert now, da noch kein Endzeitpunkt bekannt ist. Aus der Historie von Herrn Tell kann man entnehmen, dass er bereits zwei Umzüge hinter sich hat und momentan wieder an seiner ersten Adresse wohnt. Wie in Abschnitt 3.1.3 erwähnt wurde, führt die Verwendung von Intervallen in einem solchen Fall zu Redundanz: Weil Herr Tell mit einem Unterbruch an der Käshaldenstrasse 3 wohnt, benötigt man, abgesehen von den Zeitangaben, zwei identische Tupel. Diese Redundanz könnte nur durch temporale Elemente vermieden werden.

Löschen. Um einen Eintrag zu löschen, setzt man seinen Endzeitpunkt. Der bereits vorhandene Eintrag wird dabei überschrieben. Dieses logische Löschen führt somit zu keinem physischen

Löschvorgang, da man sonst die Historie verlieren würde. Das Tupel über Hilde Kurz illustriert diesen Fall.

Das vorgestellte Verfahren unterscheidet sich hinsichtlich der Verwendung des Wertes *now* geringfügig von demjenigen in [ElNa]. In dieser Arbeit wird nämlich der allgemeinere Standpunkt vertreten, dass man unter Umständen schon beim Einfügen eines neuen Tupels dessen Gültigkeitsendzeitpunkt kennen kann. Man führe sich dazu den Fall einer Adressumleitung eines Zeitungsabonnements während den Ferien vor Augen. [ElNa] hingegen gehen davon aus, dass man den Endzeitpunkt nicht im Voraus kennt und ihn deshalb erst bei einer Änderung setzt. Aus diesem Grund sind in [ElNa] nur jene Tupel aktuell, die den Wert *now* aufweisen.

Um die in Abschnitt 3.2.1 erwähnten Nachteile der Tupel-Versionierung im Falle von sich ändernden Attributwerten aufzuzeigen, wird nun angenommen, dass Frau Xiang am 11. Juni 1996 Herrn Wong heiratete und den Namen ihres Gatten annahm. Leider hielt die Ehe nicht lange, so dass am 26. November 1998 die Scheidung erfolgte. Seitdem trägt Frau Wong wieder ihren Mädchennamen. Die Ehe hatte keinen Umzug zur Folge. Diese Tatsachen sind in Tabelle 3-3 ersichtlich. Das Beispiel wurde in der Absicht modifiziert, die zu besprechenden Nachteile noch augenscheinlicher zu machen.

KUNDE

<u>KdNr</u>	<u>NName</u>	<u>VName</u>	<u>Strasse</u>	<u>PLZ</u>	<u>GebDatum</u>	<u>GültigAb</u>	<u>GültigBis</u>
1	Xiang	Lian	Lotosblütenweg 7	8002	13.11.41	01.01.90	10.06.96
1	Wong	Lian	Lotosblütenweg 7	8002	13.11.41	11.06.96	25.11.98
1	Xiang	Lian	Lotosblütenweg 7	8002	13.11.41	26.11.98	31.12.99
1	Xiang	Lian	Drachenstrasse 13	8004	13.11.41	01.01.00	now
2	Eiffel	Pierre	Am Triumphbogen	8200	14.07.63	01.10.00	now
3	Tell	Köbi	Käshaldenstrasse 3	8052	01.08.35	01.01.60	30.09.74
3	Tell	Köbi	Läckerligasse 1	4051	01.08.35	01.10.74	31.12.86
3	Tell	Köbi	Käshaldenstrasse 3	8052	01.08.35	01.01.87	now
4	Kurz	Hilde	Hopfenweg 34	3011	22.06.52	01.05.91	30.09.94

Tabelle 3-3: Nachteile der Tupel-Versionierung bei Änderung der Attributwerte

Redundanz. Obwohl die Änderungen – zweimaliger Namenswechsel und Umzug – im Fall von Frau Xiang nur die Attributwerte betreffen und nicht das Tupel als Ganzes, muss bei jeder Mutation ein neuer Eintrag in die Datenbank eingefügt werden. Dies führt dazu, dass gleichbleibende Werte redundant gespeichert werden. So werden die Werte 1, Xiang, Lian, Lotosblütenweg 7, 8002 und 13.11.41 mehrmals aufgeführt. Auch bei den anderen Tupeln aus Tabelle 3-3 ist Redundanz zu verzeichnen, im Beispiel von Frau Xiang fällt diese aber gravierender aus, da die Änderungen durch asynchron ändernde Attribute provoziert wurden. Um die Gefahr von Modifikationsanomalien zu umgehen, muss diese Redundanz explizit gehandhabt werden.

Vertikale, temporale Anomalie. Das Verfahren der Tupel-Versionierung im Fall von ändernden Attributwerten führt auch dazu, dass zusammengehörende Informationen über mehrere Tupel verstreut werden. So umfasst die Historie von Frau Xiang vier Tupel. Da für das relationale Datenbankverwaltungssystem das Tupel die primäre Dateneinheit ist, betrachtet es jedes Tupel als eigenständigen Eintrag. Die Unterstützung der konzeptionellen Zusammengehörigkeit mehrerer Tupel muss durch die Anwendung, beispielsweise durch Betrachtung des ursprünglichen Primärschlüssels KdNr, erbracht werden.

Keine explizite Historie der Attributwerte. Tabelle 3-3 macht ebenfalls deutlich, dass die Historie eines Attributwertes nicht mehr explizit zu entnehmen ist: Die Frage, von wann bis wann Frau Xiang am Lotosblütenweg 7 wohnte, kann erst durch mehrere Tupelvergleiche beantwortet werden. Wie in Abschnitt 3.2.1 besprochen, könnte dieser Nachteil durch Hinzunahme einer zusätzlichen Zeitangabe auf Attributebene behoben werden. Dies wurde in Tabelle 3-4 für die als variabel angenommenen Attribute NName, Strasse und PLZ gemacht, wobei sich Strasse und PLZ die Zeitangabe teilen, da sie immer synchron ändern. Es geht nun auf einen Blick hervor, dass Frau Xiang vom 1. Januar 1990 bis zum 31. Dezember 1999 am Lotosblütenweg 7 wohnte. Obwohl dieses Vorgehen die Historie der Attributwerte nun explizit macht, ist die Historisierungseinheit immer noch das Tupel. Die Nachteile der Redundanz und der vertikalen, temporalen Anomalie bleiben somit bestehen.

KUNDE

KdNr	NName	NNGültigAb	NNGültigBis	VName	Strasse	PLZ	AdrGültigAb	AdrGültigBis	GebDatum	GültigAb	GültigBis
1	Xiang	13.11.41	10.06.96	Lian	Lotosblütenweg 7	8002	01.01.90	31.12.99	13.11.41	01.01.90	10.06.96
1	Wong	11.06.96	25.11.98	Lian	Lotosblütenweg 7	8002	01.01.90	31.12.99	13.11.41	11.06.96	25.11.98
1	Xiang	26.11.98	now	Lian	Lotosblütenweg 7	8002	01.01.90	31.12.99	13.11.41	26.11.98	31.12.99
1	Xiang	26.11.98	now	Lian	Drachenstrasse 13	8004	01.01.00	now	13.11.41	01.01.00	now
2	Eiffel	14.07.63	now	Pierre	Am Triumphbogen	8200	01.10.00	now	14.07.63	01.10.00	now
3	Tell	01.08.36	now	Köbi	Käshaldenstrasse 3	8052	01.01.60	30.09.74	01.08.35	01.01.60	30.09.74
3	Tell	01.08.36	now	Köbi	Läckerligasse 1	4051	01.10.74	31.12.86	01.08.35	01.10.74	31.12.86
3	Tell	01.08.36	now	Köbi	Käshaldenstrasse 3	8052	01.01.87	now	01.08.35	01.01.87	now
4	Kurz	22.06.52	now	Hilde	Hopfenweg 34	3011	01.05.91	30.09.94	22.06.52	01.05.91	30.09.94

Tabelle 3-4: Explizite Historie der Attributwerte

Die aufgeführten Nachteile würden nicht auftreten, wenn man im Falle von sich ändernden Attributwerten die Attribut-Versionierung verwenden würde. Wie bereits erwähnt, ist dies jedoch bei relationalen Datenbanken allgemein nicht zu empfehlen. Es sei an dieser Stelle aber noch darauf hingewiesen, dass die Attribut-Versionierung nicht das Allerheilmittel gegen Redundanz überhaupt ist. Die Redundanz, die sich aufgrund der Verwendung von Intervallen ergeben kann, würde auch bei einer Attribut-Versionierung weiterhin bestehen bleiben: So müsste im Fall von Frau Xiang ihr Name zweimal aufgeführt werden, da sie ihn während zweier, sich nicht berührender Zeitintervalle trägt. Das gleiche gilt für die Adressangabe von Herrn Tell. Völlige Redundanzfreiheit lässt sich folglich nur durch Attribut-Versionierung in Kombination mit temporalen Elementen erzielen.

Gültigkeitszeit mit Momentaufnahmen

Wie bereits in Abschnitt 3.1.3 angesprochen, ist die ereignisorientierte Modellierung eines Bankkontos einer zustandsorientierten vorzuziehen: Anstatt fortlaufend den aktuellen Kontostand zu berechnen und abzuspeichern, genügt es, sämtliche Gutschriften und Belastungen in der Datenbank zu führen. Dies wird in Tabelle 3-1 durch die Relation BUCHUNG erreicht. Da eine Buchung ohne Angabe ihres Valuta-Datums für die Bank wertlos ist, wurde die Relation BUCHUNG bereits in Tabelle 3-1 um Historisierungsaspekte, nämlich das Valuta-Datum, erweitert. Es handelt sich also bei der Relation BUCHUNG aus Tabelle 3-1 um eine temporale Relation, die auf Tupelebene historisiert und Momentaufnahmen in Gültigkeitszeit verwendet. Insofern ist die Beschriftung von Tabelle 3-1 nicht ganz korrekt. Tabelle 3-5 zeigt die Relation BUCHUNG aus Tabelle 3-1 mit zugehöriger Extension nochmals. Hierbei wurde das historisierungsspezifische Attribut Valuta in **GültigAm** umbenannt, um die Verwendung von Gültigkeitszeit explizit zu machen. Tabelle 3-5 zeigt ebenfalls die zugehörigen

Relationen KONTO und KONTOTYP. So ist beispielsweise ersichtlich, dass Frau Xiang per 1. Januar 2000 ihre Lohnzahlung von CHF 6'293.- erhielt. Ebenfalls am 1. Januar 2000 liess Frau Xiang CHF 1'000.- von ihrem Salärkonto auf ihr Sparkonto überweisen. Es ist noch anzufügen, dass in der Praxis höchstwahrscheinlich auch die Relation KONTOTYP historisiert werden würde, um die Entwicklung der Zinssätze festhalten zu können. Dies könnte über die Verwendung von Zeitintervallen in Gültigkeitszeit passieren.

KONTO

KtoNr	Typ	KdNr
100	S	1
101	E	3
102	S	4
103	P	2
104	P	1
105	S	2

KONTOTYP

Typ	Bezeichnung	Zinssatz
E	Seniorenkonto	0.020
J	Jugendkonto	0.020
K	Kontokorrent	0.005
M	Mitarbeiterkonto	0.025
P	Sparkonto	0.015
S	Salärkonto	0.010

BUCHUNG

BuchNr	Betrag	GültigAm	KtoNr
1000	6'293.00	01.01.00	100
1001	2'005.00	01.01.00	101
1002	-1'000.00	01.01.00	100
1003	1'000.00	01.01.00	104
1004	5'230.00	01.01.00	105
1005	4'560.00	01.01.00	102
1006	-800.00	02.01.00	105
1007	800.00	02.01.00	103
1008	-50.00	03.01.00	102
1009	-200.00	04.01.00	101
1010	-100.00	05.01.00	100
1011	-265.20	07.01.00	100
1012	-425.55	07.01.00	105

Tabelle 3-5: Gültigkeitszeit mit Momentaufnahmen

Da bei einer ereignisorientierten Modellierung nicht der Zustand selber, sondern die ereignisauslösende Transaktion in der Datenbank abgebildet wird, vereinfacht sich das Vorgehen bei Modifikationsoperationen beträchtlich: Es „reduziert“ sich quasi auf reine Einfügeoperationen. Einfüge- und Änderungsoperationen einer zustandsorientierten Modellierung machen sich in einer ereignisorientierten Modellierung als reine Einfügeoperationen bemerkbar. Das Führen und Nachführen von Intervallgrenzen entfällt, da man den Zustand nicht mehr explizit angibt. Das Löschen eines Kontos bewirkt, dass keine neuen Buchungen mehr erfolgen, hat aber keinen physischen Löschvorgang zur Folge.

Es ist zu beachten, dass bei einer ereignisorientierten Modellierung die Änderungen das Tupel als Ganzes betreffen. Die Tupel-Versionierung bietet sich also als adäquates Modellierungsmittel an, weshalb die in Abschnitt 3.2.1 angesprochenen Nachteile nicht auftreten. Auch die Vergabe eines neuen Primärschlüssels ist aus dem gleichen Grund hinfällig.

Gültigkeits- und Transaktionszeit mit Zeitintervallen

Die simultane Verwendung von Gültigkeits- und Transaktionszeit erhöht die Ausdrucksfähigkeit beträchtlich. Wie bereits in Abschnitt 3.1.2 erwähnt, wird dadurch ersichtlich, ob ein Eintrag retroaktiv, synchron oder proaktiv zu seiner Erfassung gültig ist. Zudem gehen Korrekturen am Datenbestand als solche hervor.

Tabelle 3-6 zeigt die Erweiterung von Tabelle 3-2 um Zeitintervalle in Transaktionszeit. Durch die zusätzliche Verwendung der Transaktionszeit wird neben der Entwicklungsgeschichte der Entität in der Realwelt auch diejenige des Datenbestandes festgehalten. Die Attribute **GültigAb** und **GültigBis** drücken weiterhin die Zeitdauer in Gültigkeitszeit aus. Neu sind die in Transaktionszeit gemessenen Attribute **Eingesetzt** und **Ersetzt**, welche die Zeitdauer angeben, während welcher der Eintrag in der Datenbank gültig war. Die Intervallgrenzen bezeichnen die Zeitpunkte des „Systemeingriffs“ und werden aus diesem Grund automatisch durch das Datenbankverwaltungssystem geliefert. Sie sind durch den Benutzer nicht editierbar. Aus heutiger Sicht gültige Einträge haben für **Ersetzt** den Wert **uc**, was für „until changed“, also „noch nicht geändert“ steht. Solche Einträge geben die Historie der Entität wieder und entsprechen somit den Gültigkeitszeit-Tupeln in Tabelle 3-2. Aktuelle Einträge müssen folglich für **Ersetzt** den Wert **uc** tragen und den Betrachtungszeitpunkt in ihrem Gültigkeitszeitintervall einschliessen. Der neue Primärschlüssel wird durch die Kombination der Attribute **KdNr**, **GültigAb** und **Eingesetzt** gebildet, da die ersten beiden Attribute nicht mehr eindeutig sind.

KUNDE

KdNr	NName	VName	Strasse	PLZ	GebDatum	GültigAb	GültigBis	Eingesetzt	Ersetzt
1	Xiang	Lian	Lotosblütenweg 7	8002	13.11.41	01.01.90	now	04.01.1990, 10:14:33	05.01.2000, 14:35:27
1	Xiang	Lian	Lotosblütenweg 7	8002	13.11.41	01.01.90	31.12.99	05.01.2000, 14:35:27	uc
1	Xiang	Lian	Drachenstrasse 13	8004	13.11.41	01.01.00	now	05.01.2000, 14:35:27	uc
2	Eiffel	Pierre	Am Triumphbogen	8200	14.07.63	01.10.00	now	01.10.2000, 09:31:08	uc
3	Tell	Köbi	Käshaldenstrasse 3	8052	01.08.35	01.01.60	now	10.01.1960, 16:54:01	13.10.1974, 07:55:13
3	Tell	Köbi	Käshaldenstrasse 3	8052	01.08.35	01.01.60	30.09.74	13.10.1974, 07:55:13	uc
3	Tell	Köbi	Läckerligasse 1	4051	01.08.35	01.10.74	now	13.10.1974, 07:55:13	07.01.1987, 15:02:00
3	Tell	Köbi	Läckerligasse 1	4051	01.08.35	01.10.74	31.12.86	07.01.1987, 15:02:00	uc
3	Tell	Köbi	Käshaldenstrasse 3	8052	01.08.35	01.01.87	now	07.01.1987, 15:02:00	uc
4	Kurz	Hilde	Hopfenweg 34	3011	22.06.52	01.05.91	now	30.04.1991, 17:33:04	30.10.1994, 08:04:02
4	Kurz	Hilde	Hopfenweg 34	3011	22.06.52	01.05.91	30.09.94	30.10.1994, 08:04:02	uc

Tabelle 3-6: Gültigkeits- und Transaktionszeit mit Zeitintervallen

Folgend wird der Einfluss der Historisierung mit Gültigkeitszeit und Transaktionszeit auf die Modifikationsoperationen diskutiert. Die vorgestellten Verfahren basieren wiederum auf denjenigen in [ElNa], wobei, wie bereits weiter oben erwähnt, der Wert *now* leicht anders gehandhabt wird:

Einfügen. Wird ein neuer Eintrag in die Datenbank eingefügt, so muss, wie in Abschnitt „Gültigkeitszeit mit Zeitintervallen“, Seite 22 diskutiert, die Gültigkeitszeit entsprechend durch den Benutzer angegeben werden. Zusätzlich misst das System den Zeitpunkt der Einfügeoperation und trägt diesen Wert unter *Eingesetzt* ein. *Ersetzt* erhält den Wert *uc*, um die Gültigkeit des Eintrags anzuzeigen. Das Tupel Pierre Eiffel verdeutlicht diesen Vorgang.

Ändern. Änderungsoperationen fallen bei simultaner Verwendung von Gültigkeits- und Transaktionszeit mit Zeitintervallen komplexer aus, will man die volle Ausdrucksfähigkeit dieser Modellierung aufrecht erhalten. Damit für jeden vergangenen Zeitpunkt nachvollziehbar ist, was damals in der Datenbank eingetragen war, muss bei Änderungsoperationen folgende Vorgehensweise gewählt werden. Hierbei steht *g* für den Zeitpunkt, ab welchem der neue Eintrag aktuell ist, und *g*-sinngemäss für denjenigen, bis zu welchem der zu ändernde Eintrag aktuell war. Der Zeitpunkt der Änderungsoperation wird durch *t* symbolisiert.

1. Das zu modifizierende Tupel wird zweimal kopiert. Beide Kopien werden zusätzlich in die Datenbank aufgenommen.
2. In der ersten Kopie wird *GültigBis* mit *g*- überschrieben. *Eingesetzt* erhält den Wert *t*, *Ersetzt* den Wert *uc*.
3. Erst in der zweiten Kopie werden die zu ändernden Attributwerte entsprechend mutiert. Die Attribute *GültigAb* und *GültigBis* werden gesetzt, wobei *GültigAb* den Wert *g* und *GültigBis* den bereits bekannten Endzeitpunkt bzw. bei Unwissenheit den Wert *now* erhält. *Eingesetzt* weist den Wert *t* auf, *Ersetzt* den Wert *uc*.
4. Im ursprünglichen, zu modifizierenden Tupel wird *Ersetzt* auf *t* abgeändert.

Dieses Vorgehen wird am Beispiel von Frau Xiang erläutert. Vor dem Umzug steht folgendes Tupel in der Datenbank:

<u>KdNr</u>	<u>NName</u>	<u>VName</u>	<u>Strasse</u>	<u>PLZ</u>	<u>GebDatum</u>	<u>GültigAb</u>	<u>GültigBis</u>	<u>Eingesetzt</u>	<u>Ersetzt</u>
1	Xiang	Lian	Lotosblütenweg 7	8002	13.11.41	01.01.90	now	04.01.1990, 10:14:33	uc

Dieses wird zweimal kopiert. In der ersten Kopie wird die Adresse Lotosblütenweg 7 deaktiviert, indem *GültigBis* den Wert 31.12.1999 erhält. *Eingesetzt* markiert den Änderungszeitpunkt mit 05.01.2000, 14:35:27. Das Resultat ist im zweiten Tupel in Tabelle 3-6 ersichtlich. Die zweite Kopie soll nun zum aktuellen Eintrag werden. Hierzu werden die Attribute *Strasse* und *PLZ* aufdatiert. Da Frau Xiang fortan auf unbestimmte Zeitdauer an der Drachenstrasse 13 wohnt, steht unter *GültigAb* 01.01.2000, unter *GültigBis* *now* und unter *Ersetzt* *uc*. *Eingesetzt* enthält wiederum den Änderungszeitpunkt 05.01.2000, 14:35:27. Dies geht aus dem dritten Tupel in Tabelle 3-6 hervor. Das ursprüngliche Tupel wird schliesslich ungültig gesetzt, indem *Ersetzt* von *uc* auf 05.01.2000, 14:35:27 abgeändert wird, was im ersten Eintrag in Tabelle 3-6 ersichtlich ist. Dieses Invalidieren ist deshalb notwendig, weil die Information, dass Frau Xiang zum heutigen Zeitpunkt noch immer (*now*!) am Lotosblütenweg 7 wohne, nicht mehr der Realwelt entspricht. Das soeben illustrierte Vorgehen kann in analoger Weise auf Köbi Tell angewendet werden.

Auffällig an diesem Verfahren ist sicher einmal, dass bei jeder Änderung immer zwei neue Tupel generiert werden und nicht nur eines, was auf den ersten Blick naheliegender erscheinen würde. Dies ist die Konsequenz davon, dass man bei einem bestehenden Tupel keine Attributwerte ändern darf, ausser es handle sich um das Attribut *Ersetzt* mit dem Wert *uc*. Der Grund dieser Vorgehensweise wird weiter unten erläutert werden.

Löschen. Um einen Eintrag zu löschen, wird zuerst eine Kopie des aktuellen Eintrags zusätzlich in die Datenbank eingefügt. Der aktuelle Eintrag wird dann ungültig gemacht, indem Ersetzt von uc auf den aktuellen Transaktionszeitpunkt abgeändert wird. In der Kopie wird schliesslich GültigBis mit dem entsprechenden Datum überschrieben. Eingesetzt erhält den Transaktionszeitpunkt als Wert, Ersetzt weist den Wert uc vor, um das Tupel als gültig zu erklären. Ein Beispiel dafür bilden die zwei Tupel über Hilde Kurz in Tabelle 3-6.

Es ist festzuhalten, dass dieses logische Löschen kein physisches Löschen zur Folge hat, ansonsten ja die Historisierung unmöglich gemacht würde. Da, wie bereits erwähnt, in einem aktuellen Tupel abgesehen von Ersetzt keine Änderungen an bestehenden Attributwerten gemacht werden dürfen, führt ein Löschvorgang zu einem zusätzlichen Eintrag in der Datenbank. Näheres dazu folgt weiter unten. Zunächst sollen die in Abschnitt 3.1.2 erwähnten Vorteile einer simultanen Verwendung von Gültigkeitszeit und Transaktionszeit am Beispiel illustriert werden:

Zeitenfolge von Gültigkeits- und Transaktionszeit. Die simultane Verwendung von Gültigkeitszeit und Transaktionszeit erlaubt die zeitliche Abfolge, wann die Information in der Realwelt gültig ist und wann sie in der Datenbank verfügbar ist, zu beschreiben. Es geht somit klar hervor, ob ein Eintrag **retroaktiv**, **synchron** oder **proaktiv** zu seiner Erfassung gültig ist. So kann man beispielsweise aus Tabelle 3-6 entnehmen, dass der Umzug von Frau Xiang an die Drachenstrasse 13 per 01.01.2000 retroaktiv am 05.01.2000 um 14:35:27 in der Datenbank eingetragen wurde. Der Eintrag über Pierre Eiffel wurde synchron erfasst. Hilde Kurz illustriert schliesslich den proaktiven Fall.

Korrekturen. Nur bei simultaner Verwendung von Gültigkeitszeit und Transaktionszeit werden Korrekturen am Datenbestand als solche ersichtlich. Zur Illustration nehmen wir an, dass Frau Xiangs Adresse am 05.01.2000 fälschlicherweise als Drachenstrasse 12 eingetragen wurde. Der Irrtum wurde erst am 22.02.2000 entdeckt und behoben. Wie Tabelle 3-7 zeigt, wird das zu korrigierende Tupel zuerst einmal kopiert und eingefügt und dann durch Setzen des Transaktionsendzeitpunktes als ungültig erklärt. In der Kopie kann dann die Korrektur vorgenommen werden. Eingesetzt erhält den Transaktionszeitpunkt, Ersetzt den Wert uc. Dieses Beispiel veranschaulicht nochmals die Bedeutung des Wertes uc: Wie bereits erwähnt, markiert er Tupel, die zum Betrachtungszeitpunkt bis auf weiteres gültig sind. So ist es beispielsweise aus heutiger Sicht richtig, dass Frau Xiang seit dem 1. Januar 2000 an der Drachenstrasse 13 wohnt. Korrekt ist ebenfalls, dass Frau Xiang vom 01.01.1990 bis zum 31.12.1999 am Lotosblütenweg 7 wohnte. Aus der Tabelle 3-7 geht aber ebenfalls hervor, dass man vom 5. Januar bis zum 22. Februar 2000 der Meinung war, dass Frau Xiang an der Drachenstrasse 12 wohne. Weil diese Information aus heutiger Sicht fehlerhaft ist, trägt das Tupel für Ersetzt das Datum der Korrektur. Da sich ein korrigiertes Tupel hinsichtlich der Werte für KdNr und GültigAb nicht vom zu korrigierenden Tupel unterscheidet, führten Korrekturen bei alleiniger Verwendung von Gültigkeitszeit zur Überschreibung des zu korrigierenden Datensatzes.

KUNDE

<u>KdNr</u>	<u>NName</u>	<u>VName</u>	<u>Strasse</u>	<u>PLZ</u>	<u>GebDatum</u>	<u>GültigAb</u>	<u>GültigBis</u>	<u>Eingesetzt</u>	<u>Ersetzt</u>
1	Xiang	Lian	Lotosblütenweg 7	8002	13.11.41	01.01.90	now	04.01.1990, 10:14:33	05.01.2000, 14:35:27
1	Xiang	Lian	Lotosblütenweg 7	8002	13.11.41	01.01.90	31.12.99	05.01.2000, 14:35:27	uc
1	Xiang	Lian	Drachenstrasse 12	8002	13.11.41	01.01.00	now	05.01.2000, 14:35:27	22.02.2000, 09:34:02
1	Xiang	Lian	Drachenstrasse 13	8002	13.11.41	01.01.00	now	22.02.2000, 09:34:02	uc

Tabelle 3-7: Korrekturen bei simultaner Verwendung von Gültigkeits- und Transaktionszeit

Nach diesen Ausführungen kann noch ein weiterer, in [ElNa] nicht diskutierter Aspekt des Löschens erörtert werden. Bisweilen wurde der Fall gezeigt, bei welchem ein Tupel während einer gewissen Zeitspanne gültig war und danach gelöscht wird. Es ist aber auch der Fall denkbar, dass ein Tupel

fälschlicherweise in den Datenbestand aufgenommen wurde und deshalb nach Aufdeckung des Irrtums gelöscht wird. Durch die simultane Verwendung von Gültigkeits- und Transaktionszeit lässt sich auch dieser Fall ausdrücken, indem der Gedanke der Korrektur gewissermassen auf das Löschen übertragen wird: Das zu löschende Tupel wird lediglich durch Überschreiben von Ersetzt mit dem Endzeitpunkt invalidiert. Es kommt aber zu keinem zusätzlichen Eintrag. Das untenstehende Beispiel soll dies anhand von Frau Hilde Kurz illustrieren.

<u>KdNr</u>	<u>NName</u>	<u>VName</u>	<u>Strasse</u>	<u>PLZ</u>	<u>GebDatum</u>	<u>GültigAb</u>	<u>GültigBis</u>	<u>Eingesetzt</u>	<u>Ersetzt</u>
4	Kurz	Hilde	Hopfenweg 34	3011	22.06.52	01.05.91	now	30.04.1991, 17:33:04	30.10.1994, 08:04:02

Abschliessend soll der Grund für die spezielle Vorgehensweise des soeben illustrierten Verfahrens erläutert werden. Dadurch, dass in einem zu modifizierenden Tupel – abgesehen von Ersetzt – keine Attributwerte geändert werden, kann für jeden vergangenen Zeitpunkt der Datenbankzustand rekonstruiert werden. Sobald aber weitere Attributwerte modifiziert werden, ist dies nicht mehr gewährleistet. Stellt man sich beispielsweise vor (siehe untenstehende Tabelle), dass auch der Attributwert GültigBis verändert werden darf – was auf den ersten Blick zur Vermeidung eines zusätzlichen Tupels als sinnvoll erscheinen würde – wäre es nicht mehr ersichtlich, ob GültigBis bis zum 5. Januar 2000 den Wert now oder bereits den Endzeitpunkt 31.12.99 enthielt. Die Historie der Datenbank würde also in einem solchen Fall nicht mehr vollständig hervorgehen. Zudem wäre die Interpretation von uc nicht mehr dieselbe.

<u>KdNr</u>	<u>NName</u>	<u>VName</u>	<u>Strasse</u>	<u>PLZ</u>	<u>GebDatum</u>	<u>GültigAb</u>	<u>GültigBis</u>	<u>Eingesetzt</u>	<u>Ersetzt</u>
1	Xiang	Lian	Lotosblütenweg 7	8002	13.11.41	01.01.90	31.12.99	04.01.1990, 10:14:33	05.01.2000, 14:35:27
1	Xiang	Lian	Drachenstrasse 13	8004	13.11.41	01.01.00	now	05.01.2000, 14:35:27	uc

Die Überlegenheit des in diesem Abschnitt diskutierten Verfahrens soll noch an einem weiteren Beispiel illustriert werden, nämlich für den Fall, bei welchem eine Modifikationsoperation in den Schattenzeitraum einer anderen fällt. Dafür sei zuerst der Begriff Schattenzeitraum erläutert. Ein so genannter **Schattenzeitraum** (siehe Abbildung 3-3) entsteht bei retroaktiven und proaktiven Modifikationsoperationen und bezeichnet die Zeitspanne zwischen der Durchführung einer Änderung in der Datenbank und dem Inkrafttreten dieser Änderung in der Realwelt. Stellt man sich nun den Fall vor, dass Frau Xiang zusätzlich zu ihrem Wohnungswechsel auch noch per 3. Januar 2000 heiratet, liegt das eingangs diskutierte Szenario vor: Der Namenswechsel fällt in den Schattenzeitraum der Adressmutation. In einem solchen Fall führt allein das auf den ersten Blick als unnötig und kompliziert erscheinende Verfahren mit eingeschränkter Änderungsmöglichkeit zum Ziel und ist deshalb unbestritten die allein richtige Wahl (siehe Abbildung 3-4).

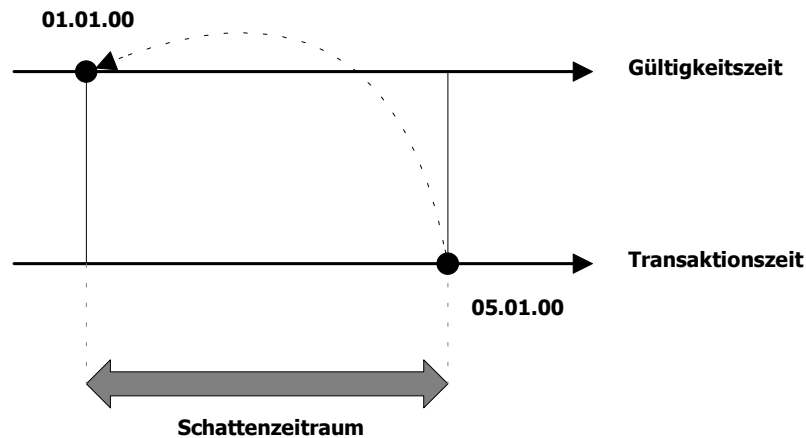


Abbildung 3-3: Schattenzeitraum bei einer retroaktiven Modifikation

	KdNr	NName	VName	Strasse	PLZ	GültigAb	GültigBis	Eingesetzt	Ersetzt
a	1	Xiang	Lian	Lotosblütenweg 7	8002	01.01.90	now	04.01.1990, 10:14:33	uc
a'	1	Xiang	Lian	Lotosblütenweg 7	8002	01.01.90	now	04.01.1990, 10:14:33	03.01.2000, 11:12:32
b	1	Xiang	Lian	Lotosblütenweg 7	8002	01.01.90	02.01.00	03.01.2000, 11:12:32	uc
c	1	Wong	Lian	Lotosblütenweg 7	8002	03.01.00	now	03.01.2000, 11:12:32	uc
a'	1	Xiang	Lian	Lotosblütenweg 7	8002	01.01.90	now	04.01.1990, 10:14:33	03.01.2000, 11:12:32
b'	1	Xiang	Lian	Lotosblütenweg 7	8002	01.01.90	02.01.00	03.01.2000, 11:12:32	05.01.2000, 14:35:27
c'	1	Wong	Lian	Lotosblütenweg 7	8002	03.01.00	now	03.01.2000, 11:12:32	05.01.2000, 14:35:27
b''	1	Xiang	Lian	Lotosblütenweg 7	8002	01.01.90	31.12.99	05.01.2000, 14:35:27	uc
d	1	Xiang	Lian	Drachenstrasse 13	8004	01.01.00	02.01.00	05.01.2000, 14:35:27	uc
c''	1	Wong	Lian	Drachenstrasse 13	8004	03.01.00	now	05.01.2000, 14:35:27	uc

Namensänderung
per 03.01.00 am
03.01.2000, 11:12:32

Umzug
per 01.01.00 am
05.01.2000, 14:35:27

Abbildung 3-4: Modifikationsoperation innerhalb eines Schattenzeitraumes⁴

Gültigkeits- und Transaktionszeit mit Momentaufnahmen

Die in Abschnitt „Gültigkeitszeit mit Momentaufnahmen“, Seite 25 gezeigte Relation BUCHUNG wird nun um die Angabe der Transaktionszeit erweitert (siehe Tabelle 3-8). Hierzu werden zusätzlich die Attribute Eingesetzt und Ersetzt aufgeführt. Es ist zu beachten, dass die Datenbankentwicklung durch Angabe der Zeitdauer modelliert wird, da der Entwicklungsprozess stufenweise konstant und zustandsorientiert ist. Das Bankkonto selbst bleibt aber durch die Relation BUCHUNG ereignisorientiert modelliert, weshalb für die Angabe der Gültigkeitszeit Zeitpunkte verwendet werden. Aufgrund der ereignisorientierten Abbildung der Entität gelten die in Abschnitt „Gültigkeitszeit mit

⁴ Aus Platzgründen wurde darauf verzichtet, das Geburtsdatum erneut anzuzeigen.

Momentaufnahmen“, Seite 25 gemachten Aussagen grundsätzlich weiterhin. Einzig KdNr bleibt nicht mehr als Primärschlüssel erhalten, sondern muss neu durch BuchNr und Eingesetzt gebildet werden.

BUCHUNG

<u>BuchNr</u>	<u>Betrag</u>	<u>GültigAm</u>	<u>KtoNr</u>	<u>Eingesetzt</u>	<u>Ersetzt</u>
1000	6'293.00	01.01.00	100	29.12.1999, 03:21:36	uc
1001	2'005.00	01.01.00	101	29.12.1999, 03:21:37	uc
1002	-1'000.00	01.01.00	100	29.12.1999, 03:21:38	uc
1003	1'000.00	01.01.00	104	29.12.1999, 03:21:39	uc
1004	5'230.00	01.01.00	105	29.12.1999, 03:21:40	uc
1005	4'560.00	01.01.00	102	29.12.1999, 03:21:41	uc
1006	-800.00	02.01.00	105	03.01.2000, 01:03:44	uc
1007	800.00	02.01.00	103	03.01.2000, 01:03:45	uc
1008	-50.00	03.01.00	102	05.01.2000, 03:22:02	06.01.2000, 22:02:21
1008	-100.00	03.01.00	102	06.01.2000, 22:02:21	uc
1009	-200.00	04.01.00	101	05.01.2000, 05:55:01	uc
1010	-100.00	05.01.00	100	06.01.2000, 00:03:49	uc
1011	-265.20	07.01.00	100	07.01.2000, 23:11:59	uc
1012	-425.55	07.01.00	105	08.01.2000, 02:48:39	uc

Tabelle 3-8: Gültigkeits- und Transaktionszeit mit Momentaufnahmen

Tabelle 3-8 zeigt die Zeitenfolge von Gültigkeits- und Transaktionszeit. So wurden beispielsweise die Buchungen 1000 bis 1005 proaktiv erfasst. Weiter illustriert Tabelle 3-8, dass Korrekturen am Datenbestand als solche hervorgehen: Dem Salärkonto von Frau Hilde Kurz wurde am 05.01.2000, 03:22:02 versehentlich CHF 50.- zu wenig belastet, was am 06.01.2000, 22:02:21 entsprechend korrigiert wurde. Dieses Beispiel macht auch deutlich, dass die explizite Angabe beider Intervallgrenzen bei der Transaktionszeit notwendig ist, ansonsten Korrekturen nicht ersichtlich würden.

Referentielle Integrität

In all jenen Fällen, in denen es zu einer Erweiterung des ursprünglichen Primärschlüssels kommt, bedarf es für die Aufrechterhaltung der **referentiellen Integrität** einer besonderen relationalen Modellierung. Als Beispiel diene die Primär-Fremdschlüssel-Beziehung zwischen der Relation KUNDE (siehe Tabelle 3-6) und KONTO (siehe Tabelle 3-5). Aufgrund der Historisierung auf Tupelebene wird der neue Primärschlüssel von KUNDE durch die Attribute KdNr, GültigAb und Eingesetzt gebildet. Folglich müsste KdNr in der Relation KONTO durch das Attribut-Tripel (KdNr, GültigAb, Eingesetzt) ersetzt werden. Dieses Vorgehen ist aber gar nicht zu empfehlen: Zum einen hätte jede Änderung eines Attributwertes in der Relation KUNDE die entsprechende Anpassung des Fremdschlüssels in der Relation KONTO zur Folge, zum anderen ist die Abgrenzung, welche Kontoinformation zu welcher Version eines Kunden-Tupels gehört, algorithmisch fast nicht umsetzbar. Zudem müssten diese Probleme auf Applikationsebene gelöst werden, da sie auf Systemebene keine Unterstützung finden. Es zeigt sich also, dass hier die Aufrechterhaltung der referentiellen Integrität äusserst kompliziert ausfällt. Viel eleganter und einfacher ist die in Abbildung 3-5 vorgeschlagene relationale

Modellierung. Dort wird der ursprüngliche Primärschlüssel KdNr der zu historisierenden Relation KUNDE in die separate Hilfsrelation H1 ausgelagert. Die Hilfsrelation H2 erlaubt dann die Zuordnung der Konten zu den einzelnen Kunden. Obwohl diese Lösung zu weiteren Joins führt, umgeht sie das Aufdatieren von Fremdschlüsseln und die zeitliche Abgrenzung von Kontoinformationen. Das in Abbildung 3-5 illustrierte Verfahren bedarf somit zur Aufrechterhaltung der referentiellen Integrität keinerlei weiteren Unterstützung auf Applikationsebene, was die zusätzlichen Joins um ein Vielfaches aufwiegt. Auch der Fall, wo die Relation KONTO ebenfalls historisiert würde, könnte auf die gleiche Weise modelliert werden: Der Primärschlüssel KtoNr von KONTO würden dann analog zu KdNr in eine weitere Hilfsrelation H3 ausgelagert werden.

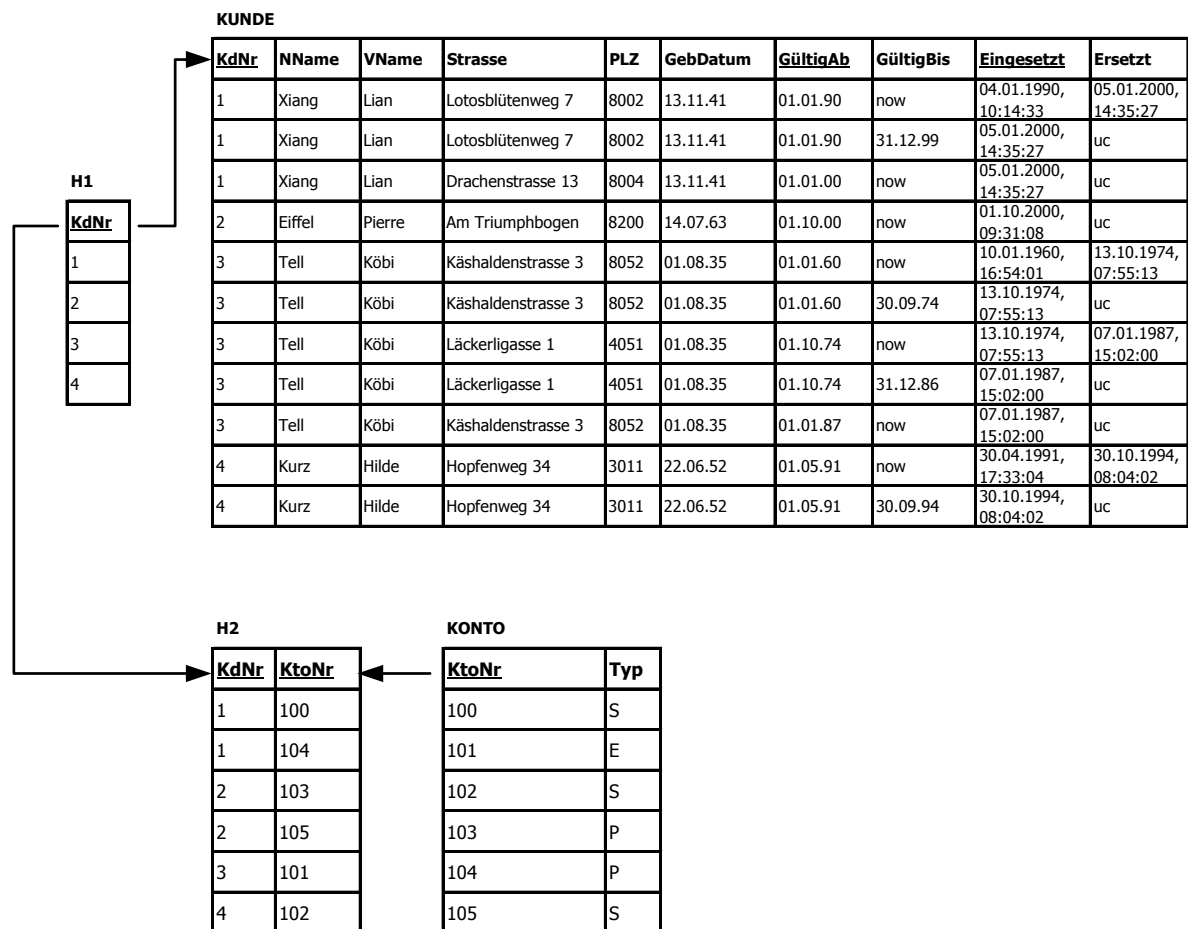


Abbildung 3-5: Referentielle Integrität bei temporaler Erweiterung

3.2.2 Temporale Erweiterung einer Datenbank mit komplexen Strukturen

Sobald ein Datenbanksystem **komplexe Strukturen** wie beispielsweise mehrwertige Attribute erlaubt, sind einem hinsichtlich der Historisierung nicht mehr die Hände gebunden: Die Attribut-Versionierung bietet sich dann als weiteres Historisierungskonzept an. In die Kategorie solcher Datenbanken fallen objektrelationale und objektorientierte Datenbanken, aber auch alle Erweiterungen relationaler Datenbanken, die sich nicht mehr an die erste Normalform halten. Dadurch, dass beide Historisierungskonzepte gleichermassen zur Verfügung stehen, kann man sie auch sinngemäss einsetzen: die Attribut-Versionierung im Fall von ändernden Attributwerten, die Tupel-Versionierung im Fall von ändernden Tupeln. Die weiter oben erwähnten Nachteile relationaler Datenbanken treten somit nicht mehr auf: Man ist nicht mehr gezwungen, auf Tupelebene zu historisieren, auch wenn die

Gegebenheiten die Attribut-Versionierung erfordern. Insofern sind komplex strukturierte Datenbanken für die Historisierung besser geeignet.

Da sich der in dieser Arbeit zu entwickelnde Prototyp auf relationale Datenbanken beschränkt, wird an dieser Stelle auf Beispiele zur temporalen Erweiterung von Datenbanken mit komplexen Strukturen verzichtet.

3.3 HISTORISIERUNGSASPEKTE IM MULTIDIMENSIONALEN DATENMODELL

Wie bereits in Abschnitt 2.3 erwähnt, ist beim multidimensionalen Datenmodell die Zeit ein integraler Bestandteil und wird als separate Dimension formuliert. Dadurch wird es möglich, die zeitliche Entwicklung der interessierenden Fakten festzuhalten. Durch die multidimensionale Datenmodellierung wird somit lediglich eine Historisierung der Fakten erzielt, die Dimensionen werden gewissermassen als statisch betrachtet.

Obwohl es grundsätzlich zutrifft, dass Fakten im Gegensatz zu Dimensionen äusserst dynamisch sind, wäre aber die Annahme verfehlt, dass Dimensionen nie ändern. Änderungen von Dimensionselementen kommen vor und erfordern ihre Berücksichtigung. Da Dimensionen anders als Fakten nicht direkt mit der Zeitdimension gekoppelt sind und somit nicht bereits durch die Art und Weise der Modellierung historisiert werden, erfordert die Historisierung von Dimensionselementen eine spezielle Vorgehensweise. Vorschläge dazu finden sich in der Literatur unter dem Begriff der so genannten **Slowly Changing Dimensions**. Dieser Begriff wurde durch Ralph Kimball [Kimb1] geprägt und soll den relativ statischen Charakter von Dimensionen im Vergleich zu Fakten verdeutlichen und dadurch ihre Nichtberücksichtigung hinsichtlich der Historisierung im multidimensionalen Modell rechtfertigen. Es ist aber anzufügen, dass es durchaus auch Dimensionen gibt, deren Elemente häufig ändern.

In den folgenden Abschnitten wird gezeigt, wie das Problem der Slowly Changing Dimensions angegangen werden kann: Abschnitt 3.3.1 diskutiert die Historisierung im aus Kapitel 2 bekannten Star-Schema, und Abschnitt 3.3.2 zeigt, wie die aus den temporalen Datenbanken gewonnenen Erkenntnisse in das Star-Schema eingebracht werden können. Die Arbeit beschränkt sich also wiederum aufgrund bekannter Gründe auf die Darstellung des relationalen Falls.

3.3.1 Herkömmliches Star-Schema

Wie bereits einführend erwähnt, geschieht im multidimensionalen Datenmodell die Historisierung der Fakten durch die Art der Modellierung. Die Historisierung der Dimensionen hingegen muss separat behandelt werden. Im Folgenden wird die Historisierung anhand des bereits bekannten Star-Schemas aufgezeigt. Grundsätzlich lässt sich festhalten, dass aufgrund der relationalen Modellierung auf **Tupelebene** historisiert wird.

Historisierung von Fakten

Im Star-Schema wird die zeitliche Entwicklung der Fakten festgehalten, indem jedes neue Faktum als weiterer Eintrag in die Fakten-Relation eingeht. Die einzelnen Fakten werden durch Primär-Fremdschlüssel-Beziehungen an ihre entsprechende Zeitangabe gekoppelt. Im Gegensatz zu den bisweilen besprochenen Beispielen erfolgt hier also die **Zeitangabe** nicht explizit, sondern **implizit** über eine Primär-Fremdschlüssel-Beziehung. Meistens wird die **Gültigkeitszeit** allein verwendet. Die Transaktionszeit könnte aber grundsätzlich als zweite Zeitdimension modelliert werden. Scheinbar plötzliche Veränderungen im Datenwürfel zwischen zwei Betrachtungszeitpunkten sind dann als Folge einer retroaktiven Mutation oder einer Korrektur erklärbar. Ausschlaggebend für die Hinzunahme der

Transaktionszeit ist aber, ob der dadurch erzielte Informationsgewinn für die Datenanalyse auf dieser hohen Abstraktionsstufe relevant ist. Speziell zu erwähnen ist die Tatsache, dass zur Modellierung des Entwicklungsprozesses lediglich **Momentaufnahmen** zur Verfügung stehen: Jedem Faktum wird über die Primär-Fremdschlüssel-Beziehung ein bestimmter Zeitpunkt der Zeitdimension zugeordnet. Diese Einschränkung auf Momentaufnahmen verhindert eine zustandsorientierte Modellierung eines stufenweise konstanten Entwicklungsprozesses. Eine Annäherung an die zustandsorientierte Modellierung kann durch die Verwendung von Zeitreihen geschehen, was aber mit einem Informationsverlust und möglicherweise Redundanz einhergehen würde (siehe Abschnitt 3.3.2).

Historisierung von Dimensionen

Für jedes Data Warehouse muss die Entscheidung gefällt werden, wie man mit möglichen Änderungen der Dimensionselemente verfährt – ob man sie historisiert und, wenn ja, wie. Ralph Kimball hat dazu die folgenden drei **Verfahrens-Typen für den Umgang mit Slowly Changing Dimensions** definiert [Kimb1], [Kimb2], [Kimb3]:

- ❖ **Typ 1:** Änderungen von Dimensionselementen werden nicht historisiert. Die alten Werte werden einfach mit den neuen **überschrieben**.
- ❖ **Typ 2:** Jede Änderung eines Dimensionselementes führt zur Generierung eines **neuen Tupels**, das zusätzlich in die Dimensions-Relation eingefügt wird. Dieses Vorgehen erfordert die Erzeugung und Verwaltung von neuen Primärschlüsseln.
- ❖ **Typ 3:** Neue Werte werden in einem **“Current“-Feld** geführt, der unmittelbar vorhergehende Wert bleibt im ursprünglichen Feld erhalten.

Typ 1: Überschreiben. Verfahren vom Typ 1 machen deutlich, dass von Fall zu Fall abgewogen werden muss, ob eine Historisierung sinnvoll ist oder ob man unter Umständen mit dem Informationsverlust leben kann. Für die Gewinnung neuer Kenntnisse hinsichtlich der Historisierung ist Typ 1 somit nicht weiter interessant.

Typ 2: Generierung eines neuen Tupels. Bei diesem Verfahren wird nach jeder Änderung ein neues Tupel erzeugt und in die bestehende Dimensions-Relation eingefügt. Die **Zeitangabe** erfolgt **implizit**, indem dem Tupel über eine Primär-Fremdschlüssel-Beziehung das zugehörige Faktum zugeordnet wird, welches wiederum indirekt den Zeitbezug über die Verknüpfung mit der Zeitrelation herstellt (siehe Abbildung 3-6). Die Zeitangabe erstreckt sich somit über **zwei Indirektionsstufen** und ist an die Existenz des zugehörigen Faktums gebunden: Liegt nämlich für eine Version eines Dimensionselementes kein Faktum vor, geht der Zeitbezug verloren. Im Prinzip liegt bei diesem Vorgehen der Schwerpunkt der Historisierung weiterhin auf den Fakten: Die Historie der Dimensionen geht nur dort eindeutig hervor, wo sie in Verbindung zu den Fakten steht. Zudem wird aufgrund ihrer Abhängigkeit von der Fakten-Relation die Historie einer Dimension lediglich auf **Zeitpunkte** abgebildet, was die Angabe der Zustandsdauer unmöglich macht. Wenn man aber davon ausgeht, dass in einem multidimensionalen Datenmodell die Fakten primäres Augenmerk sind, sind diese Nachteile nicht weiter relevant. Ansonsten liefert das in Abschnitt 3.3.2 vorgestellte temporale Star-Schema einen möglichen Lösungsansatz.

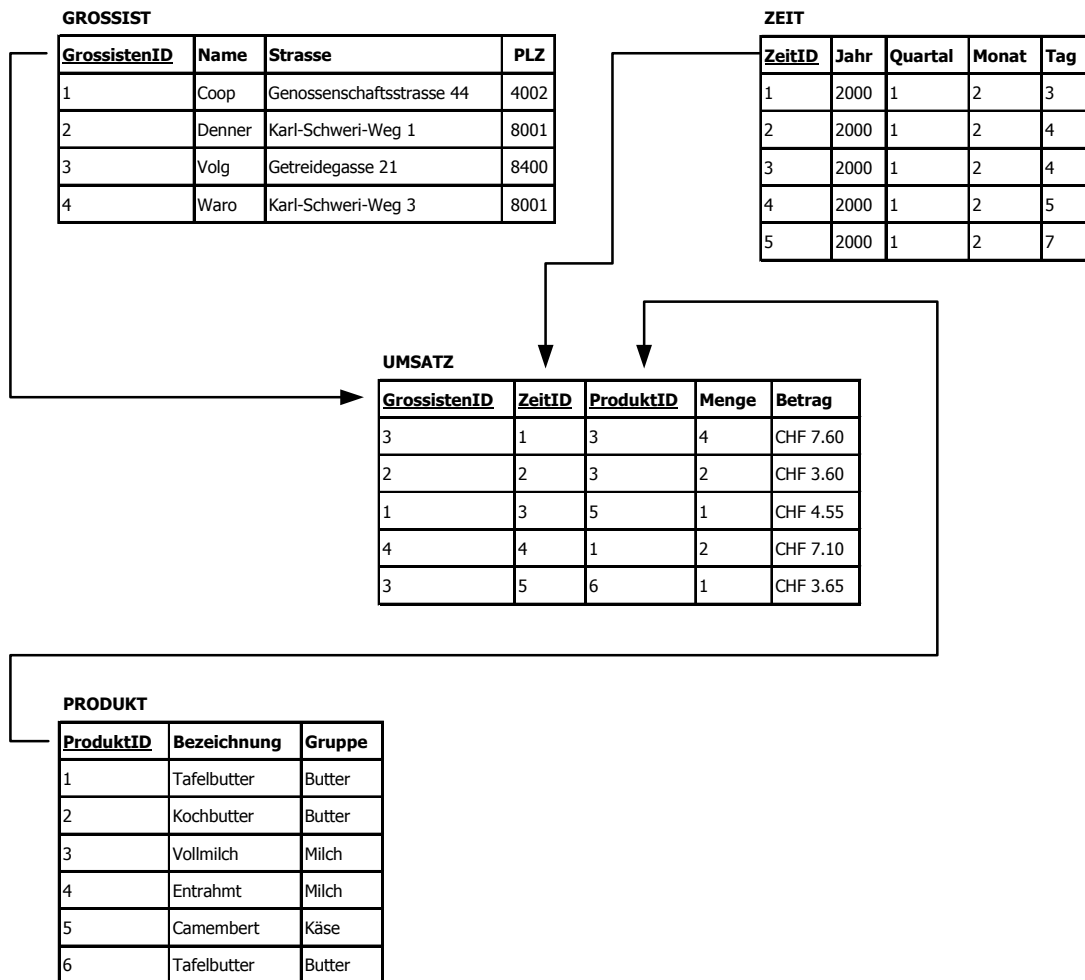


Abbildung 3-6: Extension zum Star-Schema in Abbildung 2-4

Aufgrund der impliziten Zeitangabe entfällt die aus Abschnitt 3.2.1 bekannte Möglichkeit, Zeitpunkte für den Primärschlüssel zu verwenden. Löst man das Problem der Slowly Changing Dimensions nach Typ 2, ist man folglich gezwungen, für jeden neuen Eintrag einen **neuen Primärschlüssel** zu generieren (siehe Tafelbutter in Abbildung 3-6). Dabei macht [Kimb2] die Forderung geltend, künstlich erzeugte Schlüsselstellvertreter (engl. surrogate key) zu verwenden, wobei er eine fortlaufende, ganzzahlige Nummerierung vorschlägt. Damit aber ein Eintrag in der Dimensions-Relation weiterhin erkennen lässt, um welches Dimensionselement es sich handelt, müssen die neuen Primärschlüssel auf die ursprünglichen Schlüssel der Quellen abgebildet werden. Der Aufwand für die Schlüsselgenerierung und –verwaltung ist deshalb nicht zu unterschätzen.

Falls die Änderungen in den Dimensionen von den Attributwerten ausgehen – was meistens auch der Fall ist –, führt die Tupel-Versionierung zur temporalen, vertikalen Anomalie und somit zur Verwendung verschiedener Primärschlüssel für unterschiedliche Versionen desselben Dimensionselementes. Dies wiederum bewirkt, dass die **Fakten-Historie** hinsichtlich der Versionen eines Dimensionselementes **partitioniert** wird: Jede Partition umfasst jene Fakten, die denselben Fremdschlüssel haben, die sich also auf dieselbe Version eines Dimensionselementes beziehen. Die Zusammengehörigkeit mehrerer Elementversionen und dadurch die Zusammenfassung mehrerer Partitionen zu einer Historie kann durch die Modellierung nicht ausgedrückt werden.

In [Kimb1], [Kimb2] wird die Verwendung von Zeitangaben in Dimensionen für die Angabe der Zustandsdauer oder für den Fall, dass ein Dimensionselement ohne zugehöriges Faktum in den Datenbestand eingeht, diskutiert. Es wird aber der Standpunkt vertreten, dass Mittelpunkt der Analyse

immer die Fakten sind und deshalb eine implizite Zeitangabe über zwei Stufen und somit die Verwendung von Momentaufnahmen ausreichend ist. Zudem ist der Fall, dass für eine Version eines Dimensionselementes kein Faktum vorliegt, eher selten. Weiter wird darauf hingewiesen, dass man bei der Interpretation von Zeitangaben für Dimensionen vorsichtig sein sollte. So wird aufgezeigt, dass die Zeitangaben von Dimensionen und Fakten einen unterschiedlichen Bezug haben: Bei Dimensionen beziehen sich die Zeitangaben auf die Entwicklungsgeschichte der Dimensionselemente, bei Fakten auf die Entwicklungsgeschichte der Fakten selber. Nimmt man beispielsweise an, dass der Milchprodukte-Produzent Moloko aus Kapitel 2 die Zusammensetzung des Schokoladejoghurts geringfügig ändert ohne eine neue Artikelnummer zu vergeben, dann hat dies zur Folge, dass während einer gewissen Übergangszeit in den Läden sowohl die alten als auch die neuen Schokoladejoghurts verkauft werden. Aus dem Vergleich des Verkaufsdatums mit den Intervallgrenzen lässt sich demzufolge nicht ableiten, ob es sich bei dem Verkauf um ein Joghurt nach neuer oder nach alter Rezeptur handelt. Für diese Information benötigt man die Verknüpfung des Dimensionselementes mit dem entsprechenden Faktum. Dieses Problem stellt sich immer dann, wenn sich Partitionen der Fakten-Historie während einer Übergangszeit überlappen. Schliesslich wird auch das mit der Tupel-Versionierung verbundene Problem der nicht expliziten Attributwert-Historie erwähnt.

Schliesslich sei noch erwähnt, dass es das vorgestellte Verfahren nicht erlaubt, die Fakten-Historie mit alten Werten fortzuführen bzw. die Vergangenheit mit den neuen Werten zu betrachten. Das Paradebeispiel hierfür wäre die Neubenennung von Verkaufsgebieten und der Wunsch, dass man die neuen Verkaufszahlen zusätzlich auch noch den alten Gebietsnamen zuordnen kann. Diese Möglichkeit entfällt aufgrund der angesprochenen Partitionierung der Fakten-Historie.

Typ 3: Führen eines „Current“-Feldes. Das zuletzt angesprochene Problem, dass Typ 2 das **Fortschreiben** der **Fakten-Historie** mit alten Werten bzw. das **Rückschreiben** mit neuen Werten nicht erlaubt, stellt sich beim Verfahren nach Typ 3 nicht mehr. Dadurch, dass die Aufdatierung im „Current“-Attribut passiert und somit keine Neugenerierung eines Tupels zur Folge hat, kommt es auch nicht zur temporalen, vertikalen Anomalie: Sämtliche Informationen über ein und dasselbe Dimensionselement sind im gleichen Tupel enthalten. Die Fakten-Historie für ein und dasselbe Dimensionselement wird also hinsichtlich der einzelnen Versionen nicht partitioniert, da ein Fremdschlüssel auf alle verfügbaren Versionen eines Dimensionselementes verweist. Es ist somit möglich, die Fakten entweder in Bezug zu den alten Angaben oder den neuen Angaben zu setzen. Weil es aber gerade zu keiner Partitionierung kommt, ist auch nicht mehr ersichtlich, wann ein neuer Wert das erste Mal in Zusammenhang mit einem Faktum „auftaucht“. Um diese zeitliche Zuordnung aber dennoch möglich zu machen, wird in [Kimb1] die Verwendung einer Zeitangabe für das „Current“-Feld empfohlen.

Während bei den bisherigen Beispielen eine Relation durch die Einfügungen neuer Tupel dynamisch wuchs, erfordert dieses Vorgehen eine vorangehende statische Anpassung des Relationenschemas. Die Historisierung macht sich nicht mehr durch eine Zunahme an Tupeln, sondern durch eine Zunahme an Spalten bemerkbar. In diesem Sinn führt hier die Historisierung zu einem **horizontalen Wachstum** auf Schemaebene, während sie in den bisherigen Beispielen ein vertikales Wachstum der Extension zur Folge hatte. Es ist aber festzuhalten, dass ein solches horizontales Wachstum im Unterschied zum vertikalen Wachstum durch die statische Schemadefinition beschränkt ist: Im Verfahren nach Typ 3 bleibt neben dem aktuellen Wert nur der unmittelbar vorhergehende erhalten, da das Schema nur zwei Attribute für die Historisierung zur Verfügung stellt. Ältere Werte gehen somit verloren, weshalb die **Dimensions-Historie** äusserst **lückenhaft** ausfällt.

3.3.2 Temporales Star-Schema

In diesem Abschnitt wird das **temporale Star-Schema** nach [BJSS] skizziert. Dieses versucht, einen Grossteil der in Abschnitt 3.3.1 angesprochenen Mängel des herkömmlichen Star-Schemas zu

beheben, indem es jenes um Konzepte aus den temporalen Datenbanken erweitert. Konkret nimmt sich das temporale Star-Schema folgender Mängel des herkömmlichen Star-Schemas an:

- ❖ **Keine zustandsorientierte Modellierung der Fakten-Relation:** Aufgrund der erwähnten Einschränkung auf Momentaufnahmen für die Entwicklungsmodellierung von Fakten ist eine zustandsorientierte Modellierung eines stufenweise konstanten Entwicklungsprozesses nicht möglich. Natürlich kann man einer zustandsorientierten Modellierung nahe kommen, indem man die Fakten periodisch erfasst und dadurch **Zeitreihen** zur Darstellung verwendet. Die Qualität einer Zeitreihe ist aber ganz entscheidend von der Erfassungsfrequenz abhängig: Ist sie zu hoch, führt dies zu **Redundanz**, ist sie zu niedrig, geht **Information** verloren.
- ❖ **Keine zustandsorientierte Modellierung der Dimensions-Relation bei Typ 2:** Auch die Entwicklungsmodellierung einer Dimensions-Relation nach Typ 2 ist durch ihre Bindung an die Fakten-Relation auf Momentaufnahmen beschränkt. Dadurch kann aber bei einem stufenweise konstanten Entwicklungsprozess der Dimensionselemente die Zustandsdauer nicht angegeben werden. Im Allgemeinen könnte man die ungefähre Zustandsdauer aus den Zeitangaben der zugehörigen Fakten ableiten. Für den Fall, dass sich Partitionen der Fakten-Historie zeitweilig überlappen, aber keine neue Artikelnummer vergeben wurde, ist dies nicht mehr möglich. Fragen der Art „Wie viele Schokoladdejoghurts alter Rezeptur wurden nach der Einführung des neuen Schokoladdejoghurts noch verkauft?“ können also nicht beantwortet werden.
- ❖ **Eventueller Verlust des Zeitbezugs von Dimensionselementen bei Typ 2:** Geht unter Umständen eine Version eines Dimensionselementes ohne zugehöriges Faktum in den Datenbestand ein, geht bei Typ 2 der Zeitbezug verloren.
- ❖ **Schlüsselgenerierung bei Typ 2:** Die Tupel-Versionierung bei Dimensionen und die implizite Zeitangabe führen bei der Historisierung nach Typ 2 dazu, dass neue Primärschlüssel generiert und verwaltet werden müssen. Dies ist mit einem beträchtlichen Aufwand verbunden.

Grundidee des temporalen Star-Schemas ist, die Zeit nicht mehr als Dimension zu modellieren, sondern sie jeweils explizit in die entsprechenden Tupel einzufügen. Dadurch entfällt die Bindung der Fakten-Relation an die Zeitdimension bzw. die zeitliche Abhängigkeit einer Dimension von der Fakten-Relation und somit die Einschränkung auf Momentaufnahmen. Die explizite Zeitangabe erlaubt also die Verwendung der ganzen, aus Abschnitt 3.1.3 bekannten Palette zur Modellierung des Entwicklungsprozesses. Insbesondere ist die zustandsorientierte Modellierung eines stufenweise konstanten Entwicklungsprozesses möglich. Die ersten zwei oben aufgeführten Mängel sind somit behoben. Durch die explizite Zeitangabe wird auch Punkt drei hinfällig. Das Problem der Schlüsselgenerierung stellt sich ebenfalls nicht mehr: Wie bereits in Abschnitt 3.2.1 erwähnt, kann der ursprüngliche Primärschlüssel in Kombination mit der Zeitangabe als neuer Primärschlüssel verwendet werden.

Natürlich wird auch im temporalen Star-Schema die Verknüpfung von Dimensionen und Fakten weiterhin über eine Primär-Fremdschlüssel-Beziehung realisiert. Neu ist lediglich, dass nun auch die Dimensionen Zeitangaben aufweisen und somit an Unabhängigkeit gewinnen. Nebst der Tatsache, dass das temporale Star-Schema eine akkuratere Modellierung erlaubt, realisiert es gewissermassen auch die „Gleichstellung“ zwischen Fakten und Dimensionen hinsichtlich der Historisierung.

Abschliessend ist noch darauf hinzuweisen, dass Joins in temporalen Star-Schemas aufwändiger ausfallen als in herkömmlichen Star-Schemas. Dies hat seinen Grund darin, dass sich im temporalen Star-Schema die Primärschlüssel über zwei bzw. drei Attribute erstrecken. Weiterführende Angaben dazu sind [BJSS] zu entnehmen.

3.4 HISTORISIERUNG VON METADATEN

Bisweilen wurde die Historisierung lediglich anhand von den eigentlichen Daten illustriert. Obwohl die Grenze zwischen Daten und Metadaten nicht immer klar gezogen werden kann, ist es unbestreitbar, dass auch **Metadaten** historisiert werden müssen. Nur so kann ein ganzheitliches Bild der Entwicklungsgeschichte gezeichnet werden. Gerade im Umfeld von analytischen Datenbanken, deren Güte von einer umfassenden Metadaten-Haltung abhängt, ist die Historisierung von Metadaten äusserst brisant.

Da Metadaten selbst wiederum als „Daten“ in der Datenbank gespeichert werden und somit die gleichen Konzepte zur Darstellung Anwendung finden, unterscheidet sich die Historisierung von Metadaten nicht weiter von derjenigen der Daten selbst. Es ist aber anzufügen, dass Änderungen auf Metadatenebene auch Änderungen auf Datenebene nach sich ziehen können. So bewirkt beispielsweise die Erweiterung eines Relationenschemas um ein zusätzliches Attribut die entsprechende Anpassung der Daten. Hierbei wird aufgrund der Historisierung für jedes aktuelle Tupel ein entsprechend erweitertes eingefügt. Je nachdem wo man die Grenze zwischen Daten und Metadaten zieht, ist aber auch der umgekehrte Fall denkbar: Änderungen in den Daten können zu Änderungen in den Metadaten führen. So kann beispielsweise die Änderung eines Dimensionselementes im herkömmlichen Star-Schema die Anpassung eines Konsolidierungspfades zur Folge haben. Die Historisierung läuft somit auf beiden Ebenen gleichzeitig ab, wobei Interdependenzen zwischen diesen Ebenen bestehen. Auf eine weitere Erörterung der Interdependenz zwischen Daten und Metadaten wird jedoch verzichtet, da sie den Rahmen dieser Arbeit sprengen würde.

3.5 ABSTRAKTES HISTORISIERUNGSMODELL

Die aus den vorangehenden Abschnitten gewonnenen Einblicke in die Historisierung sollen nun gewissermassen unter einen Hut gebracht werden. Im vorliegenden Abschnitt wird versucht, das Gemeinsame aller vorgestellten Methoden herauszufiltern, um ein abstraktes Historisierungsmodell zu entwickeln. Dabei gilt als Ziel, das zu generierende Historisierungsmodell möglichst implementierungsfrei zu machen, um ein breites Anwendungsfeld zu ermöglichen. Das in diesem Abschnitt erarbeitete Historisierungsmodell bildet dann die Grundlage für die in Kapitel 4 erläuterte metadatengesteuerte Historisierung.

3.5.1 Dimensionen der Historisierung

Die bisherigen Einblicke in die Historisierung legen nahe, die Historisierung unter verschiedenen Gesichtspunkten zu betrachten: Beispielsweise muss man sich überlegen, welches die Granularitätsstufe der Historisierung ist, welche Zeitdimension verwendet und wie der Entwicklungsprozess modelliert werden soll. Die Historisierung hat also viele Facetten. Diese Facetten sollen unabhängig voneinander betrachtet und auch uneingeschränkt miteinander kombiniert werden können. Aufgrund der geforderten Orthogonalität werden in dieser Arbeit diese Gesichtspunkte bzw. Facetten wiederum als **Dimensionen** bezeichnet. Jede Dimension erlaubt eine spezielle Sichtweise auf die Historisierung, alle Dimensionen zusammen spannen gewissermassen einen Historisierungsraum auf. In diesem Raum stellt jeder Punkt eine spezifische Kombinationsmöglichkeit wie beispielsweise die Historisierung auf Tupelebene mit Gültigkeitszeit und Momentaufnahmen dar. Es gilt nun, sämtliche Dimensionen des Historisierungsmodells zu finden.

Zeitangabe

Da die Historisierung zum Ziel hat, die zeitliche Entwicklungsgeschichte der Datenbasis festzuhalten, kann auf die **Angabe der Zeit** nicht verzichtet werden. Dabei kann die Zeitangabe entweder **explizit** oder **implizit** erfolgen. Abschnitt 3.2.1 illustriert den expliziten Fall, in welchem die Tupel einer relationalen Datenbank um temporale Attribute erweitert werden. Das herkömmliche Star-Schema aus Abschnitt 3.3.1 zeigt die Möglichkeit der impliziten Zeitangabe, bei welcher der Zeitbezug indirekt über eine Primär-Fremdschlüssel-Beziehung realisiert wird. Es sei an dieser Stelle jedoch festgehalten, dass die soeben erwähnten Beispiele nicht die einzigen Repräsentanten für eine explizite bzw. implizite Zeitangabe sind. Andere Implementierungsformen sind durchaus denkbar. Zudem wird die Implementierung auch durch das zugrunde liegende Datenmodell bestimmt. Die Dimension Zeitangabe des abstrakten Historisierungsmodells hält somit lediglich fest, dass eine Zeitangabe erfolgen muss und dass dies entweder explizit oder implizit geschehen kann. Implementierungsspezifische Angaben werden auf dieser hohen Abstraktionsstufe bewusst unterlassen.

Zeitgranulat

Ungeachtet dessen, dass man in digitalen Rechnern zur Verwendung einer diskreten Zeitrepräsentation gezwungen ist (siehe Abschnitt 3.1.1), braucht es zur Historisierung **verschiedene Zeitgranulate**, die sich hierarchisch zu einem **Kalender** aggregieren lassen. So lassen sich beispielsweise Minuten zu Stunden, Stunden zu Tagen, Tage zu Wochen usw. zusammenfassen. Dadurch kann die Historisierung auf unterschiedlichen Resolutionsstufen ablaufen. Das zu wählende Zeitgranulat ist grundsätzlich vom zu modellierenden Sachverhalt abhängig: Die Entwicklungsgeschichte der Datenbasis wird üblicherweise in einer höheren Resolution als diejenige der Entität festgehalten. Aber auch unterhalb der Entitäten gibt es Unterschiede: Personenbezogene Angaben werden wahrscheinlich nur auf Tagesbasis genau erfasst, während Buchungstransaktionen einer Bank auf die einzelne Sekunde genau angegeben werden. Zudem ist die Bandbreite der zu verwendenden Zeitgranulate vom Verwendungszweck der Datenbank abhängig: Während in operativen Datenbanken vor allem detaillierte Daten im Vordergrund stehen, finden sich in analytischen Datenbanken sowohl detaillierte als auch aggregierte Daten, weshalb dort die ganze Palette an verfügbaren Zeitgranulaten ausgeschöpft wird.

Zeitdimensionen

Dieser Aspekt des abstrakten Historisierungsmodells macht deutlich, dass sich Zeitangaben auf unterschiedliche Sachverhalte beziehen können. So werden Zeitangaben in **Gültigkeitszeit** dazu verwendet, Sachverhalte der Realwelt wiederzugeben, während die **Transaktionszeit** zur Beschreibung des Datenbankzustands eingesetzt wird. Wie bereits in Abschnitt 3.1.2 erwähnt, stehen solche Zeitdimensionen orthogonal zueinander und können separat oder miteinander kombiniert verwendet werden. Deshalb macht es Sinn, sie als eigenständige Dimensionen des abstrakten Historisierungsmodells zu sehen. Insofern handelt es sich bei der „Dimension“ Zeitdimensionen um ein eigentliches Dimensionsbündel.

Es sei an dieser Stelle darauf hingewiesen, dass es unter Umständen noch weitere, für die Historisierung in Betracht zu ziehende Zeitdimensionen gibt. In [Ste] wird beispielsweise dargelegt, dass es Sinn machen kann, eine Beobachterperspektive einzubringen. In diesem Fall geben dann Zeitangaben an, wann man aus Beobachtersicht glaubt, dass ein Sachverhalt gültig ist. Es wird somit eine etwaige Unsicherheit ausgedrückt. Grundsätzlich gilt aber für alle solchen Zeitdimensionen, dass sie orthogonal zueinander sein müssen. Da sich Gültigkeits- und Transaktionszeit in der Literatur

weitgehend durchgesetzt haben, beschränkt sich die vorliegende Arbeit auf diese zwei Zeitdimensionen.

Datengranulat

Das Datengranulat bezeichnet jene Dateneinheit, die aufgrund der Historisierung mit jeder Änderung neu gebildet und zusätzlich in die Datenbank eingefügt wird. In Abschnitt 3.1.4 wurden die Datengranulate **Attribut** und **Tupel** diskutiert, wobei Tupel als generischer Begriff verstanden werden soll. Wie in Abschnitt 3.4 erwähnt, können auch Metadaten historisiert werden. Da auf Datenbankebene sowohl für die Daten als auch für die Metadaten dieselben Darstellungskonzepte verwendet werden, erfolgt in der Dimension Datengranulat keine weitere Unterscheidung hinsichtlich Daten und Metadaten: Der Begriff Datengranulat bezeichnet lediglich die Dateneinheit der Historisierung, ob dahinter nun Daten oder Metadaten stehen.

Obwohl sich die Arbeit bisher auf die Attribut- und Tupel-Versionierung beschränkt hat, sind weitere Granulate denkbar. So kann man beispielsweise auch **Extensionen** historisieren. Dies könnte gerade bei Metadaten durchaus sinnvoll sein. Sämtliche möglichen Datengranulate werden durch das zugrunde liegende Datenmodell bestimmt: Jede konzeptionelle Einheit im Datenmodell ist zugleich ein potentiellles Datengranulat. So bieten sich für das relationale Datenmodell die Attribut-, Tupel- und Extensions-Versionierung an.

Es sei nochmals darauf hingewiesen, dass das Datengranulat nicht unbedingt auch die Einheit sein muss, auf die sich eine Zeitangabe bezieht. Obwohl dies zwar allgemein so zutrifft, wurde in Abschnitt „Gültigkeitszeit mit Zeitintervallen“, Seite 25 der Fall der Tupel-Versionierung gezeigt, in welchem zusätzlich noch Zeitangaben auf Attributebene gesetzt wurden, um die Historie der Attributwerte explizit zu machen.

Wie bereits in Abschnitt 3.1.4 erwähnt, sollte sich das Datengranulat am Ursprung der Änderungen orientieren: Es sollte jene Dateneinheit als Datengranulat gewählt werden, von welcher die Änderung ausgeht. Das Beispiel des relationalen Datenmodells zeigt aber, dass dies nicht immer möglich ist. Dort kann aufgrund der faktischen Einschränkung auf die Tupel-Versionierung nicht immer adäquat modelliert werden, weshalb mit den besagten Nachteilen zu rechnen ist.

Entwicklungsbeschreibung

Aus Abschnitt 3.3.1 geht hervor, dass die Historisierung auch durchaus nur **lückenhaft** passieren kann. Wird nämlich im herkömmlichen Star-Schema das Problem der Slowly Changing Dimensions nach Typ 3 gelöst, entsteht eine solche lückenhafte Historie: nur die letzten n (im Beispiel zwei) Werte werden beibehalten. Bei einer **nahtlosen** Entwicklungsbeschreibung hingegen werden sämtliche vergangenen Werte in der Datenbasis festgehalten.

Entwicklungsprozess

Die Modellierung eines Entwicklungsprozesses kann entweder durch **Momentaufnahmen** oder durch die Angabe der **Zeitdauer** erfolgen. Hierbei ist die Auswahl zwischen diesen beiden Konzepten vom zugrunde liegenden Entwicklungsprozess und der gewählten Abbildung einer Entität in die Datenbank abhängig. Dies wurde in Abschnitt 3.1.3 bereits ausführlich erläutert.

Wachstumsrichtung

Das in Abschnitt 3.3.1 erwähnte Verfahren nach Typ 3 zur Handhabung von Slowly Changing Dimensions verlangt eine weitere Dimension, die mit dem Namen **Wachstumsrichtung** bezeichnet wird. Sie illustriert, dass es neben der Möglichkeit, die Datenbank dynamisch zu erweitern, auch die Möglichkeit gibt, statische Veränderungen am Datenbankschema vorzunehmen und so die Einfügungen neuer Werte zu erlauben. Bildlich gesprochen führt erstere Vorgehensweise dazu, dass die Datenbank **vertikal** anwächst, letztere dazu, dass sie **horizontal** anwächst. Es ist aber darauf hinzuweisen, dass bei einer horizontalen Wachstumsrichtung dem Wachstum Grenzen gesetzt sind: Es können nur so viele neue Einträge aufgenommen werden, wie durch die Schemaänderung Freiraum geschaffen wurde. Eine horizontale Wachstumsrichtung geht somit in den meisten Fällen mit einer lückenhaften Historisierung einher.

3.5.2 Anwendung des abstrakten Historisierungsmodells

Die im vorhergehenden Abschnitt erarbeiteten Dimensionen bilden die Konstituierenden des abstrakten Historisierungsmodells (siehe Abbildung 3-7): Die Historisierung läuft entlang diesen Achsen, d.h. es muss für jede Historisierung entschieden werden, ob die Zeitangabe explizit oder implizit erfolgt, welches das zugrunde liegende Zeitgranulat für die Zeitangabe ist, welche Zeitdimension(en) verwendet wird (werden), auf welcher Granularitätsstufe historisiert wird, ob die Entwicklungsbeschreibung lückenhaft oder nahtlos ausfallen soll und ob der Entwicklungsprozess mit Momentaufnahmen oder durch Angabe der Zeitdauer modelliert werden soll. Jede Dimension muss also berücksichtigt werden.

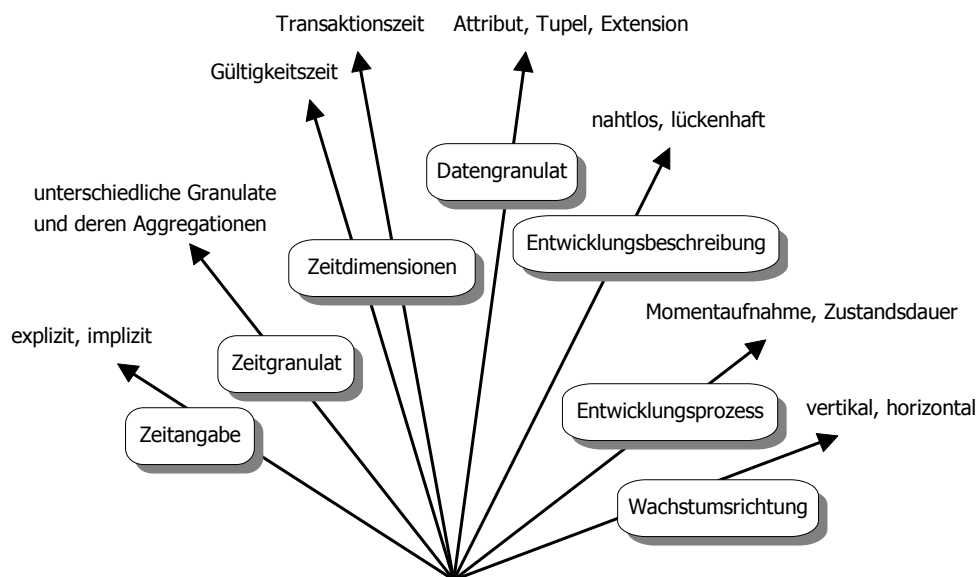


Abbildung 3-7: Abstraktes Historisierungsmodell

Wie bereits erwähnt, sind die Dimensionen des abstrakten Historisierungsmodells **orthogonal** zueinander. Grundsätzlich sind alle Dimensionselemente frei miteinander kombinierbar, jeder Punkt in dem durch die Dimensionen aufgespannten Raum ist also möglich. Erst implementierungsspezifische Faktoren wie das zugrunde liegende Datenmodell oder die Art des Entwicklungsprozesses und dessen Abbildung auf die Datenbank engen die Auswahl ein und erlauben nur noch gewisse Punkte im Historisierungsraum.

Der Schaffung eines solchen abstrakten Historisierungsmodells liegt die Idee zugrunde, dass man fortan die Historisierung auf einer implementierungsunabhängigen Ebene handhaben kann. Dadurch wird das Modell uneingeschränkt anwendbar: Mit den vorgestellten Dimensionen lässt sich sowohl die Historisierung in einer relationalen Datenbank wie auch in einer objektorientierten Datenbank bewerkstelligen. Aufgrund dieser **Universalität** bildet das abstrakte Historisierungsmodell auch die Grundlage für die in Kapitel 4 diskutierte metadatengesteuerte Historisierung. Andererseits dokumentiert dieses abstrakte Historisierungsmodell auch eine gewisse Vorgehensweise zur Historisierung, die einen sicher zum Ziel führt.

Abschliessend sei noch vermerkt, dass nicht der Anspruch auf Vollständigkeit erhoben wird: Es ist nicht gesagt, dass es sich bei den in Abbildung 3-7 gezeigten Dimensionen um sämtliche Konstituierenden des abstrakten Historisierungsmodells handelt. Da das Modell durch Beobachtung des Gegebenen und Herleitung der Gemeinsamkeiten entstanden ist, ist eine gewisse Orientierung an konkreten Implementierungen nicht von der Hand zu weisen. Dies wiederum könnte die Universalität des Modells in Frage stellen. Erst die Zukunft wird zeigen, wie es um die Universalität dieses abstrakten Historisierungsmodells bestellt ist und ob Weiterentwicklungen in der Datenbanktechnologie zur Hinzunahme weiterer Dimensionen zwingen. Dieser Punkt illustriert gerade einen weiteren Vorteil des abstrakten Historisierungsmodells, nämlich seine **Flexibilität**. Da die Dimensionen zueinander orthogonal sind, kann man uneingeschränkt weitere Dimensionen definieren und hinzufügen oder bestehende Dimensionen löschen. Aus heutiger Sicht machen die in Abbildung 3-7 aufgeführten Dimensionen aber alle Sinn und scheinen die im mathematischen Sinn minimale Menge von Erzeugenden zu sein, weshalb für weitere Betrachtungen das in Abbildung 3-7 gezeigte abstrakte Historisierungsmodell als korrekt vorausgesetzt werden soll.

4 METADATENGESTEUERTE HISTORISIERUNG IM DATA WAREHOUSING

Ausgehend von dem im vorhergehenden Kapitel erarbeiteten abstrakten Historisierungsmodell wird im vorliegenden Kapitel die metadatengesteuerte Historisierung für die Aktualisierung von Data Warehouses eingehender betrachtet.

Die bedeutende Stellung von Metadaten im Data Warehousing wurde bereits in Abschnitt 2.4 betont. Dort wurde ebenfalls erwähnt, dass Metadaten sowohl passiv zur Systemdokumentation als auch aktiv als Steuerungsinformationen für so genannte **metadatengesteuerte Werkzeuge** verwendet werden können. Da viele Prozesse im Data Warehousing heutzutage durch Werkzeuge unterstützt werden, liegt es im Rahmen einer umfassenden Metadatenverwaltung nahe, bestimmte Steuerungsinformationen ebenfalls als Metadaten im Repository abzulegen. Solche Steuerungsinformationen können statischer als auch dynamischer Natur sein. Beispiele für den ersten Fall sind Strukturdefinitionen und Konfigurationsspezifikationen. Bei dynamischen Steuerungsinformationen hingegen handelt es sich um Programmfragmente wie Parameter, Bedingungen oder Methoden. Dadurch, dass bei einem metadatengesteuerten Werkzeug Steuerungsinformationen als Metadaten im Repository abgelegt werden, ist die Programmlogik nicht mehr allein im Werkzeug verborgen, sondern wird auf das Werkzeug und das Repository verteilt. Erst zur Laufzeit entsteht der ausführbare Programmcode, indem das Werkzeug, wie in Abbildung 2-5 dargestellt, auf das Repository zugreift und die benötigten Programmfragmente dynamisch einbindet. Die Struktur des Repository, also das Metadatenschema, ist hierbei ausschlaggebend. Sie muss derart ausgestaltet sein, dass sie einerseits eine gewisse Offenheit wahrt, um möglichst vielen Werkzeugen den Zugriff zu erlauben, andererseits muss sie für jedes einzelne dieser Werkzeuge das Format der Steuerungsinformationen aufs Genaueste spezifizieren. Die Vorteile eines solchen Ansatzes sind zahlreich:

- ❖ **Flexibilität:** Solange sich ein metadatengesteuertes Werkzeug an die Spezifikation der Steuerungsinformationen durch das Repository-Schema hält, kann es beliebige Instanzen davon bearbeiten.
- ❖ **Wartbarkeit:** Mit der oben aufgeführten Flexibilität geht auch die Wartbarkeit einher: Schemainstanzen können unabhängig vom zugreifenden Werkzeug geändert werden. Dadurch wird es möglich, Steuerungsinformationen zu ändern, ohne dass dies eine Anpassung des Werkzeugs zur Folge hat. Zudem kann eine solche Änderung auch durch jemanden erfolgen, der keine Programmierkenntnisse hat.
- ❖ **Wiederverwendbarkeit:** Dadurch, dass Steuerungsinformationen aus dem privaten Bereich eines Werkzeugs ausgelagert und offen in einem Repository zugänglich gemacht werden, können sie auch von anderen Werkzeugen wiederverwendet werden.
- ❖ **Förderung einer umfassenden Metadatenverwaltung:** Indem bei einem metadatengesteuerten Werkzeug die Metadaten getrennt vom Werkzeug gespeichert und verwaltet werden, lassen sie sich ohne weiteres in ein unternehmensweites Konzept zur Verwaltung von Metadaten eingliedern. Insofern fördern sie eine umfassende Metadatenverwaltung.

Das vorliegende und das folgende Kapitel versuchen nun, ein metadatengesteuertes Werkzeug für die Historisierung von Data Warehouses zu entwickeln. Während in Kapitel 5 der Schwerpunkt auf der Implementierung eines solchen Historisierungswerkzeugs liegt, übernimmt dieses Kapitel gewissermaßen die konzeptuelle Vorarbeit: Abschnitt 4.1 erläutert die Einbettung eines solchen Werkzeugs ins Data-Warehouse-System und Abschnitt 4.2 erarbeitet ein mögliches Metadatenschema. Hierbei dient als Grundlage das aus Abschnitt 3.5 bekannte abstrakte Historisierungsmodell, um das Repository-Schema möglichst universell zu formulieren und somit ein breites Einsatzspektrum des

Historisierungswerkzeugs zu gewährleisten. In Abschnitt 4.3 wird schliesslich eine mögliche Architektur für ein metadatengesteuertes Historisierungs-System vorgestellt.

4.1 EINEBETTUNG INS DATA-WAREHOUSE-SYSTEM

Abbildung 2-1 macht deutlich, dass die Historisierung als Teil des ETL-Prozesses während der **Datenakquisition** passiert. Die in den Akquisitionsschritten modifizierten Quelldaten bilden gewissermassen den Input in das Historisierungswerkzeug, dessen Output dann ins Data Warehouse geladen wird. Somit kann das Historisierungswerkzeug nicht losgelöst von den links und rechts von ihm gelagerten Prozessen betrachtet werden.

Einerseits vereinfachen die vor- und nachgelagerten Akquisitionsprozesse die Historisierungsaufgabe, indem beispielsweise im Transformationsschritt ein einheitliches Datenformat geschaffen wird. Andererseits können sie aber auch einen einschränkenden Einfluss ausüben. So bestimmt der Extraktionsprozess massgeblich, welche Punkte des durch die Dimensionen des abstrakten Historisierungsmodells aufgespannten Historisierungsraums realisierbar sind. Beispielsweise ist bei einer benutzergesteuerten oder periodischen Extraktion nicht gewährleistet, dass die Entwicklungsbeschreibung nahtlos ausfällt, da man unter Umständen Änderungen verpasst. In diesen Zusammenhang reiht sich auch die Diskussion um den so genannten **Nettoeffekt** und dessen Handhabung durch den Extraktionsprozess ein. Unter Berücksichtigung des Nettoeffekts werden mehrere Modifikationsoperationen zusammengefasst und nur der nach ihrer Ausführung resultierende Endzustand der Datenquelle ins Data Warehouse übertragen. Zwischenzustände gehen somit dem Data Warehouse verloren.

Aber auch die Informationsfülle der Datenquellen bestimmt die Güte der Historisierung massgeblich: So setzt die Verwendung der Gültigkeitszeit auch voraus, dass der Benutzer im Quellsystem die jeweiligen Zeitangaben macht. Ist dies nicht der Fall, muss man sich mit einer Näherungslösung behelfen, indem man den Modifikationszeitpunkt für die Gültigkeitszeitangaben verwendet. In diesem Zusammenhang stellt sich auch die Frage nach der Handhabung von Korrekturen bei simultaner Verwendung von Gültigkeitszeit und Transaktionszeit (siehe Abschnitt „Gültigkeits- und Transaktionszeit mit Zeitintervallen“, Seite 29). Will man in einem solchen Fall verhindern, dass Korrekturen ebenfalls historisiert werden, müsste dies bei der Erfassung in den Quellsystemen so deklariert werden. Hierfür bietet sich beispielsweise die Verwendung eines Mutationscodes an, der dem Benutzer die Angabe erlaubt, ob es sich bei einer Mutation um eine Änderung oder um eine Korrektur handelt.

Es zeigt sich somit, dass man für die Implementierung eines Historisierungswerkzeugs die Schnittstellen links und rechts davon genau spezifizieren muss. Es muss klar festgelegt werden, welches Datenmodell für die zu historisierenden Daten verwendet wird, welche Historisierungsmöglichkeiten realisiert werden können und mit welchen Informationen aus den Quellen man rechnen darf.

4.2 METADATENSHEMA

Dieser Abschnitt versucht, das Metadatenschema, also das Schema des Repository, zu erarbeiten. Hierbei werden lediglich die historisierungsspezifischen Aspekte berücksichtigt. In einer umfassenden Metadatenverwaltung liesse sich aber ein solches historisierungsspezifisches Schema ohne weiteres in ein globales einbringen. Grundlage für die Herleitung des Schemas ist das abstrakte Historisierungsmodell aus Abschnitt 3.5. Dabei soll das Schema derart ausgestaltet werden, dass jede Historisierungsart, also jeder Punkt im Historisierungsraum, realisierbar wird. Zudem soll es eine

gewisse Offenheit wahren und dem Historisierungswerkzeug die Unterstützung beliebiger Datenmodelle erlauben.

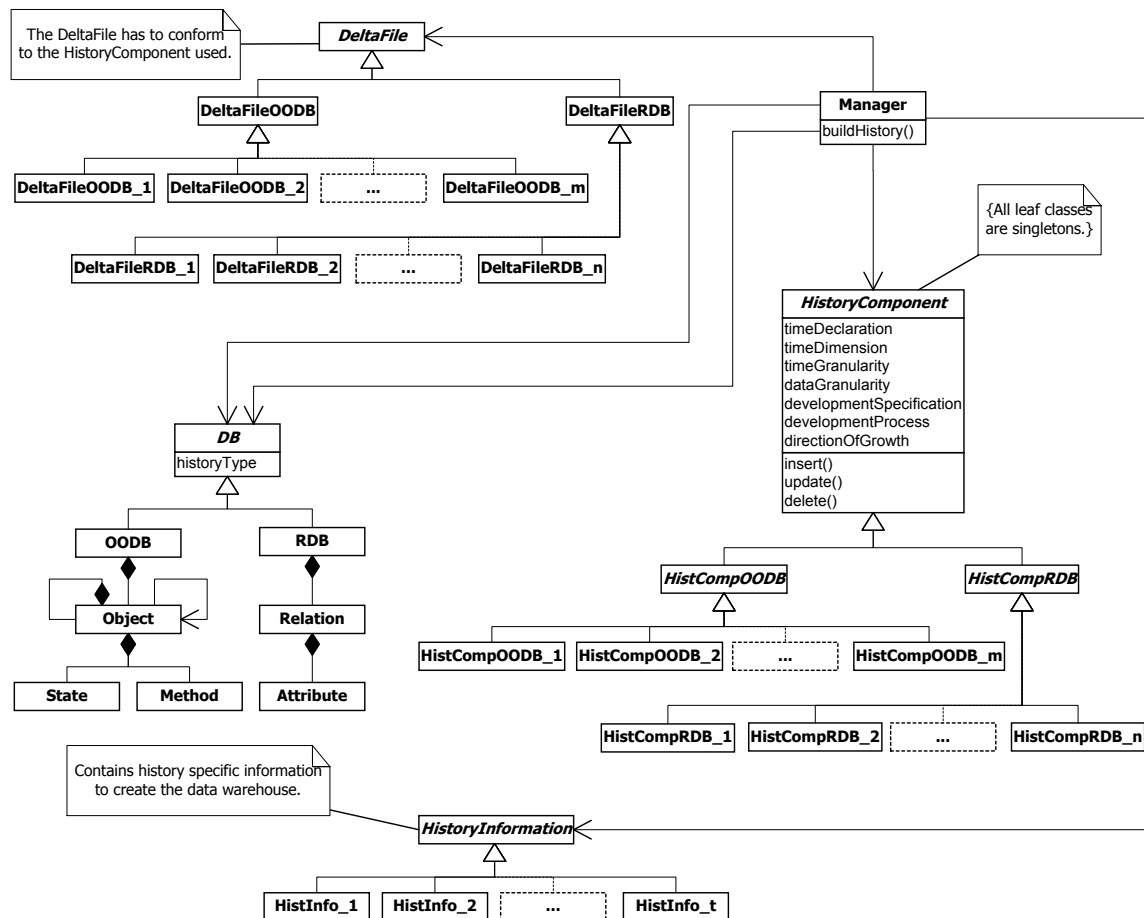


Abbildung 4-1: Metadatenschema

Abbildung 4-1 zeigt ein solches Schema. Zur Illustration wurde ein UML-Klassendiagramm [FoSc] verwendet. Wie bereits erwähnt, soll dieses Metadatenschema jegliche Historisierungsart erlauben. Es gilt also, die Dimensionen des abstrakten Historisierungsmodells ins Schema einzubringen. Dies wurde in der abstrakten Klasse **HistoryComponent** gemacht: Für jede Dimension des Historisierungsmodells definiert sie ein entsprechendes Attribut. Eine **HistoryComponent** ist dabei jene Komponente, welche die Historisierung durchführt. Dafür deklariert sie die Operationen **insert()**, **update()** und **delete()**.

Aufgrund der Tatsache, dass auch das dem Data Warehouse zugrunde liegende Datenmodell die Historisierung beeinflusst – einerseits erfordert jedes Datenmodell eine andere Implementierung der Operationen **insert()**, **update()** und **delete()**, andererseits ist nicht, wie das Beispiel des relationalen Modells zeigt, jeder Punkt im Historisierungsraum auch realisierbar (siehe Abschnitt 3.2.1) –, fand im Metadatenschema das Datenmodell ebenfalls Berücksichtigung: Für jedes Datenmodell wird eine entsprechende, wiederum abstrakte Unterklasse von **HistoryComponent** definiert. In Abbildung 4-1 wurde dies lediglich für den objektorientierten (**HistCompOODB**) und den relationalen (**HistCompRDB**) Fall gemacht. Weitere Unterklassen lassen sich aber nach Bedarf einfach hinzufügen.

Eine solche datenmodellspezifische Klasse weist nun für jeden realisierbaren Punkt im Historisierungsraum eine eigene Unterklasse auf (**HistCompOODB_1** bis **HistCompOODB_m** und

HistCompRDB_1 bis **HistCompRDB_n**). In Abbildung 4-1 wurden diese Unterklassen nummeriert, um anzudeuten, dass es sich quasi um einzelne Kombinationsmöglichkeiten des abstrakten Historisierungsmodells handelt. Die Anzahl sämtlicher Unterklassen einer jeweiligen datenmodellspezifischen Klasse ist nach oben durch sämtliche Kombinationsmöglichkeiten der Dimensionselemente des abstrakten Historisierungsmodells beschränkt. Es ist aber festzuhalten, dass aufgrund ihrer Abhängigkeit vom zugrunde liegenden Datenmodell unterschiedliche datenmodellspezifische Klassen nicht die gleiche Anzahl an Unterklassen aufweisen müssen, was in der Abbildung durch **HistCompOODB_m** und **HistCompRDB_n** verdeutlicht wird. Zudem darf die Nummerierung nicht als absolut verstanden werden. Es ist also nicht so, dass **HistCompOODB_7** den gleichen Punkt im Historisierungsraum wie **HistCompRDB_7** realisiert, da ein Datenmodell nicht alle Historisierungsarten unterstützen muss. Jede Unterklasse einer datenmodellspezifischen Klasse hat dann für die von **HistoryComponent** ererbten Attribute die entsprechenden Werte zu setzen. Zudem muss jede Unterklasse die in **HistoryComponent** deklarierten abstrakten Operationen **insert()**, **update()** und **delete()** implementieren. In Abbildung 4-1 wird ebenfalls die sinnvolle Bedingung gestellt, dass jede dieser Unterklassen ein Singleton [GHJV] sein muss, dass sie also nur einmal instanziiert werden darf.

Aus den bisherigen Ausführungen wird nun auch klar, weshalb die Wurzelklasse **HistoryComponent** und die datenmodellspezifischen Unterklassen **HistCompOODB** und **HistCompRDB** als abstrakt definiert wurden: Erst in den Blättern der Vererbungshierarchie können die Attributwerte effektiv angegeben und die Operationen implementiert werden. Die Oberklassen vereinbaren lediglich unterschiedliche Sichtweisen – historisierungsbezogen und datenmodellbezogen –, die dann miteinander kombiniert nach einer konkreten Auffächerung verlangen. Aus diesem Grund werden die einzelnen Blätter als Historisierungskomponenten und die ganze Vererbungshierarchie als Historisierungswerkzeug bezeichnet.

Die übrigen Klassen in Abbildung 4-1 verdeutlichen das Zusammenspiel mit dem Historisierungswerkzeug und eine etwaige Ablaufsemantik der Historisierung. Die für die Historisierung benötigten Daten bezieht das Historisierungswerkzeug von der Klassenhierarchie **DeltaFile**. Die abstrakte Wurzelklasse **DeltaFile** ist analog zu **HistoryComponent** in eine dreistufige Hierarchie gegliedert, wobei wiederum nach allen Historisierungsarten und Datenmodellen unterschieden wird. Jede Historisierungskomponente hat somit ihr Pendant auf Seite des **DeltaFile**, von welchem es die Daten erhält.

Eine weitere Klassenhierarchie in Abbildung 4-1 ist **DB**. Sie steht einerseits für die Quelldatenbank(en), andererseits für das Data Warehouse, was in der Abbildung durch die zwei Assoziationen angezeigt wird. Die abstrakte Klasse **DB** weist für jedes Datenmodell bzw. für jede Datenbank eine eigene Unterklasse auf. Diese datenbankspezifischen Unterklassen sind ihrerseits Kompositionen. Durch die Unterscheidung in Quell- und Zieldatenbank wird es möglich, unterschiedliche Datenmodelle für die Datenquellen und für das Data Warehouse einzusetzen. Neben Strukturinformationen der jeweiligen Datenbank, also deren Metadaten, gibt **DB** auch die zu verwendende bzw. die verwendete Historisierungskomponente an.

Für den Aufbau des Data Warehouse benötigt man historisierungsspezifische Informationen: So muss beispielsweise für den aus Abschnitt „Gültigkeits- und Transaktionszeit mit Zeitintervallen“, Seite 27 bekannten Fall angegeben werden, dass die zusätzlichen Attribute **GültigAb**, **GültigBis**, **Eingesetzt** und **Ersetzt** benötigt werden. Solche historisierungsspezifischen Angaben finden sich in der Klassenhierarchie **HistoryInformation**, welche für jeden Punkt im Historisierungsraum eine eigene Unterklasse definiert.

Gewissermassen als zentrale Drehscheibe kann die Klasse **Manager** angesehen werden. Sie koordiniert die Zusammenarbeit der einzelnen Klassen und überwacht die Historisierung. Dazu hat sie Assoziationen zu allen anderen Klassen und deklariert die Operation **buildHistory()**, welche die Historisierung auslöst. Die Ausgestaltung dieser Assoziationen in einer konkreten Implementierung ist

vielseitig. Sie kann von einfachen Referenzen zur Laufzeit bis hin zur Speicherung eines 5-Tupels reichen, das die gemeinsam verwendeten Instanzen sämtlicher Klassen einander zuordnet.

4.3 ARCHITEKTUR

Abschliessend soll im vorliegenden Abschnitt eine mögliche Architektur für ein metadatengesteuertes Historisierungs-System vorgestellt werden. Dabei umfasst ein solches System neben dem Data Warehouse und dem Repository sämtliche weitere Komponenten, welche zur Historisierung des Data Warehouse hinzugezogen werden und die Historisierung auch durchführen.

Der Architekturvorschlag in Abbildung 4-2 zeigt als zentrales Element das durch Software zu realisierende Historisierungswerkzeug. Zur Erfüllung seiner Aufgaben liest es zur Laufzeit Steuerungsinformationen aus dem Repository ein. Man beachte dabei, dass der Zugriff auf das Repository wiederum durch den Metadaten-Manager kontrolliert wird. Die für die Historisierung benötigten Daten werden in einer Datei, der so genannten Delta-Datei, zur Verfügung gestellt. Von dort liest das Historisierungswerkzeug die Daten zur Laufzeit ein und führt die Historie des Data Warehouse entsprechend nach. Wie in der Abbildung verdeutlicht, benötigt das Historisierungswerkzeug sowohl einen lesenden als auch einen schreibenden Zugriff auf das Data Warehouse. Die Zugriffe auf das Repository und auf die Delta-Datei sind hingegen nur lesender Natur.

Um den Zusammenhang zwischen Abbildung 4-1 und Abbildung 4-2 zu verdeutlichen und etwaige Missverständnisse auszuräumen, sei folgendes festgehalten: Die Struktur des in Abbildung 4-2 dargestellten Repository wird durch das Metadatenschema in Abbildung 4-1 definiert. Ähnliche oder gleiche Namen in den Abbildungen wurden absichtlich gewählt. Dahinter steckt der Gedanke, nicht etwa die jeweiligen Elemente in den Abbildungen einander gleichzusetzen, was nämlich falsch wäre, sondern die Zuordnung einer Komponente des Historisierungs-Systems zu ihrer beschreibenden Information zu erlauben. So finden sich Metadaten über die Delta-Datei in der Metadaten-Klasse DeltaFile und Metadaten über das Historisierungswerkzeug in der Klassenhierarchie HistoryComponent.

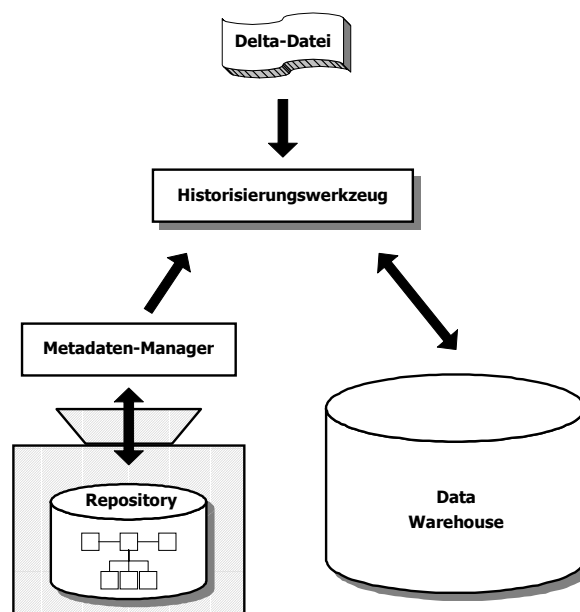


Abbildung 4-2: Architektur eines metadatengesteuerten Historisierungs-Systems

5 PROTOTYP

Das vorliegende Kapitel stellt den im Zusammenhang mit dieser Arbeit entstandenen Prototyp als eine mögliche Implementierung für ein Historisierungswerkzeug vor. Der Prototyp baut auf dem im vorangehenden Kapitel erarbeiteten Metadatenschema auf. Abschnitt 5.1 bietet einen Überblick, während Abschnitt 5.2 die aus der Implementierung gezogenen und für die dieser Arbeit zugrunde liegende Fragestellung interessanten Schlüsse darlegt.

5.1 BESCHREIBUNG

Wie bereits erwähnt, basiert der Prototyp auf dem Metadatenschema in Abbildung 4-1. Er realisiert jedoch lediglich eine Historisierungsart, welche im abstrakten Historisierungsmodell die folgenden „Koordinaten“ hat:

- ❖ **Zeitangabe:** explizit.
- ❖ **Zeitgranulat:** Datum für Gültigkeitszeit, Zeitstempel für Transaktionszeit.
- ❖ **Zeitdimensionen:** simultane Verwendung von Gültigkeitszeit und Transaktionszeit.
- ❖ **Datengranulat:** Tupel.
- ❖ **Entwicklungsbeschreibung:** nahtlos.
- ❖ **Entwicklungsprozess:** Zustandsdauer.
- ❖ **Wachstumsrichtung:** vertikal.

In den folgenden Abschnitten wird nun der Prototyp eingehend erläutert. Da der Schwerpunkt jedoch auf dem grundsätzlichen Verständnis für das Zusammenspiel der in Abbildung 4-1 dargestellten Klassen liegt und das Ganze vor allem unter dem Blickwinkel der Historisierung beleuchtet werden soll, wird auf eine ausführliche Betrachtung des Programmtextes verzichtet. Der interessierte Leser sei deshalb ermuntert, den im Anhang aufgeführten Quelltext zu konsultieren oder mit dem beigelegten Prototyp zu experimentieren.

5.1.1 Architektur und Systemangaben

Der Prototyp richtet sich grundsätzlich an der in Abbildung 4-2 dargestellten Architektur aus und umfasst die untenstehenden Elemente:

- ❖ **Java-Applikation:** Die Applikation beinhaltet einerseits die Historisierungskomponente und andererseits weitere, für die Historisierungsaufgabe benötigte Programmklassen. Sie wurde in der objektorientierten Programmiersprache Java realisiert, wobei Sun's Java Development Kit, Version 1.3.1 zum Einsatz kam.

Die Java-Applikation entspricht der Systemkomponente Historisierungswerkzeug in Abbildung 4-2.

- ❖ **Relationales Datenbanksystem:** Für die Verwaltung und Speicherung des Repository einerseits und die Verwaltung und Speicherung des Data Warehouse andererseits kam im Prototyp ein relationales Datenbanksystem zum Zug. Dazu wurde Oracle Personal Edition 8i für Windows 98, Release 2, Version 8.1.6 verwendet.

Die konzeptionell zu trennenden Systemkomponenten Data Warehouse und Repository aus Abbildung 4-2 werden also im Prototyp durch ein und dieselbe Datenbank widerspiegelt und

fallen physisch somit zusammen. Als direkte Konsequenz davon hat man im Prototyp auch auf eine Implementierung der Systemkomponente Metadaten-Manager verzichtet: Die Zugriffe auf das Repository werden durch das relationale Datenbankverwaltungssystem geregelt.

- ❖ **Testumgebung:** Die Testumgebung besteht zum einen aus einer SQL-Start-Datei, welche das Data Warehouse initialisiert, und zum anderen aus der aus Abbildung 4-2 bekannten Systemkomponente Delta-Datei, welche die zu historisierenden Daten enthält.

5.1.2 Java-Applikation

Das in Abschnitt 4.2 erarbeitete Metadatenschema kann als Ausgangslage für die Ausgestaltung der Java-Applikation dienen. Dabei ist von Fall zu Fall zu entscheiden, ob und falls ja, auf welche Art und Weise die jeweilige Metadaten-Klasse in die Implementierung einfließen soll. Insofern lässt sich das in Abbildung 4-1 dargestellte Schema auch unter einem programmiertechnischen Blickwinkel betrachten. Wie das UML-Klassendiagramm [FoSc] in Abbildung 5-1 zeigt, lassen sich die meisten Klassen des Schemas eins zu eins auf die Applikation übertragen.

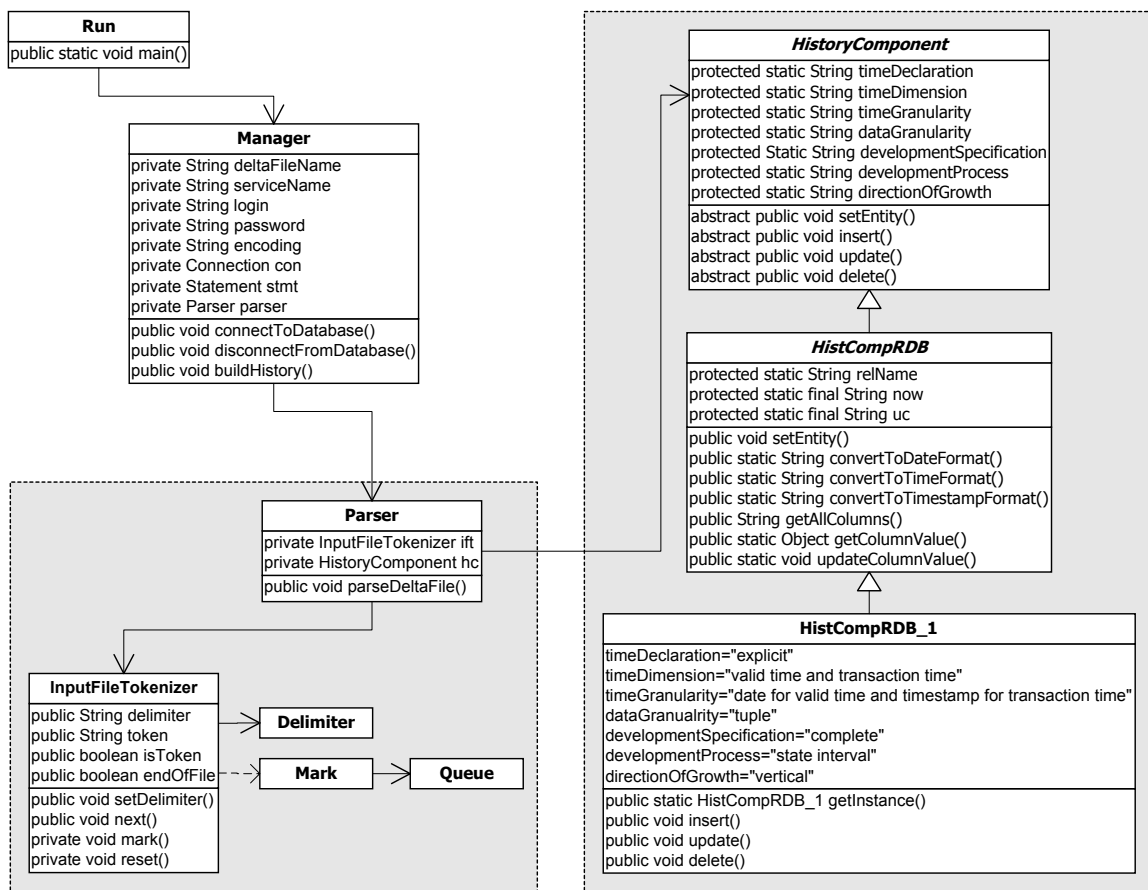


Abbildung 5-1: UML-Klassendiagramm der Java-Applikation

Abbildung 5-1 zeigt, dass im Fall des Prototyps die Programmklassen HistoryComponent, HistCompRDB, HistCompRDB_1 und Manager mit den Klassen des Metadatenschemas übereinstimmen. Dabei wurde lediglich die Historisierungskomponente HistCompRDB_1 implementiert, da der Prototyp nur die weiter oben erwähnte Historisierungsart realisiert und sich gemäss Abschnitt 5.1.1 auf den relationalen Fall beschränkt. Die übrigen Metadaten-Klassen aus Abbildung 4-1 fanden in der Java-Applikation keine Widerspiegelung als Programmklassen. Dennoch beeinflussen sie in der einen oder

anderen Weise den Programmverlauf. So muss der Java-Applikation eine Delta-Datei (**DeltaFile**) übergeben werden, welche die zu historisierenden Daten enthält. Dazu definiert die Programmklasse **Manager** die Instanzvariable **deltaFileName**, welche den Namen der Delta-Datei bezeichnet. Weiter benötigt die Historisierungskomponente in den Methoden **insert()**, **update()** und **delete()** Strukturinformationen über den Aufbau des Data Warehouse. Diese bezieht sie aus dem Repository, von der Metadaten-Klasse **DB**. Auch die Metadaten-Klasse **HistoryInformation** schlägt sich in gewisser Hinsicht in der Applikation nieder, indem sie nämlich die historisierungsspezifischen Parameter der Methoden **insert()**, **update()** und **delete()** beschreibt.

Nach diesem Überblick über das Klassendiagramm der Java-Applikation und dem Vergleich mit demjenigen der Metadaten sollen nun in groben Zügen die einzelnen Programmklassen und deren Zusammenspiel erläutert werden:

Die Klasse **Run** bildet den Einstiegspunkt in die Applikation, da sie die Klassenmethode **main()** enthält. In dieser Methode wird eine Instanz der Klasse **Manager** kreiert, an welche dann die Botschaft **buildHistory()** gesendet wird. Durch den Aufruf der Methode **buildHistory()** kommt gewissermassen die Historisierung ins Rollen.

Die Klasse **Manager** stellt zudem die Methoden **connectToDatabase()** und **disconnectFromDatabase()** zum Aufbau und Abbruch einer Verbindung zum Data Warehouse bzw. Repository zur Verfügung. Diese Methoden werden in **buildHistory()** aufgerufen: Zuerst wird mittels **connectToDatabase()** die Verbindung zur Datenbank hergestellt. Nach erfolgreicher Ausführung wird dann die Methode **parseDeltaFile()** der Klasse **Parser** aufgerufen. Dazu wurde zuvor eine Instanz der Klasse **Parser** im Konstruktor von **Manager** erstellt. Kann die Methode **parseDeltaFile()** ebenfalls fehlerfrei durchgeführt werden, wird schliesslich die Verbindung zur Datenbank abgebaut und die Applikation terminiert. Andernfalls wird die Datenbank zurückgesetzt (engl. **rollback**) und die Verbindung abgebaut.

Wie aus Abbildung 5-1 hervorgeht, bildet die Klasse **Parser** mit den Klassen **InputFileTokenizer**, **Delimiter**, **Mark** und **Queue** eine Einheit. Dabei realisiert **Parser** gewissermassen die Schnittstelle einerseits zu **Manager** und andererseits zur Historisierungskomponente: Während die Klassen **InputFileTokenizer**, **Delimiter**, **Mark** und **Queue** zur lexikalischen Verarbeitung der Delta-Datei verwendet werden, interpretiert **Parser** die resultierenden Ergebnisse und ruft eine der Methoden **insert()**, **update()** und **delete()** der Klasse **HistoryComponent** auf.

Bevor weiter auf das Zusammenspiel zwischen **Manager**, **Parser** und der Historisierungskomponente eingegangen wird, werden vorerst kurz die Klassen **InputFileTokenizer**, **Delimiter**, **Mark** und **Queue** erläutert. Dabei handelt es sich um wiederverwendbare Klassen, die es erlauben, aus einer Textdatei einzelne, durch Trennwörter (engl. **delimiter**) voneinander abgegrenzte Textsequenzen (engl. **token**) herauszufiltern. Die zu verwendenden Trennwörter werden durch den Benutzer mittels **setDelimiter()** definiert und können unterschiedlich lang sein. Dabei werden sie als separate Klasse **Delimiter** modelliert. Die lexikalische Verarbeitung wird in der Methode **next()** der Klasse **InputFileTokenizer** erreicht. Mit jedem Aufruf wird entweder die nächste Textsequenz oder das nächste Trennwort gefunden, wobei Zeilenumbrüche ignoriert, Leerzeichen aber als zum jeweiligen Text dazugehörig interpretiert werden. Je nachdem ob es sich um eine Textsequenz oder um ein Trennwort handelt, wird die boolesche Variable **isToken** dementsprechend gesetzt. Der jeweilige Inhalt ist dann in der Variable **token** bzw. **delimiter** verfügbar. Die boolesche Variable **endOfFile** kann zudem zur Iterationssteuerung verwendet werden. Da die lexikalische Analyse auf einer zeichenweisen Verarbeitung der Textdatei beruht, muss für den Aufbau der Textsequenzen bzw. der Trennwörter jeweils so lange vorausgelesen werden, bis eindeutig feststeht, ob es sich bei den eingelesenen Zeichen um eine Textsequenz oder um ein Trennwort handelt. Um dieses Vorauslesen zu erlauben, benötigt man die Klassen **Mark** und **Queue**. **Mark** widerspiegelt eine Markierung auf der einzulesenden Textdatei, welche mittels der Methode **mark()** in der Klasse **InputFileTokenizer** gesetzt werden kann. Ist eine solche Markierung einmal aktiv, werden alle fortan gelesenen Zeichen zum wiederholten Lesen auf einer Queue (Programmklasse **Queue**) abgespeichert. Die Markierung kann mittels der Methode **reset()** in der Klasse **InputFileTokenizer** aufgehoben werden, wodurch mit dem wiederholten

Lesen begonnen wird. Die Klasse `InputFileTokenizer` ist zudem so konzipiert, dass sie ineinander verschachtelte Aufrufe von `mark()` und `reset()` erlaubt.

In der Methode `parseDeltaFile()` der Klasse **Parser** wird nun also sukzessive die Methode `next()` der Klasse `InputFileTokenizer`, angewendet auf die Delta-Datei, aufgerufen. Dazu verfügt `Parser` über eine Instanz der Klasse `InputFileTokenizer`. Wie bereits angesprochen, interpretiert der `Parser` die eingelesenen Textsequenzen und Trennwörter und führt dann die Historisierung durch, indem er an die wiederum als Instanzvariable vorhandene Historisierungskomponente `hc` die entsprechende Botschaft `insert()`, `update()` oder `delete()` sendet. Bevor nun weiter auf die Klassen `HistoryComponent`, `HistCompRDB` und `HistCompRDB_1` eingegangen wird, soll nun die Klasse `Parser` und insbesondere die Methode `parseDeltaFile()` erläutert werden.

Die Methode `parseDeltaFile()` wurde so gestaltet, dass sie mit hinsichtlich Historisierungsart und Datenmodell unterschiedlichen Historisierungskomponenten zusammenarbeiten kann. Man benötigt also nicht, wie bei der Historisierungskomponente, für jede Kombination von Historisierungsart und Datenmodell einen eigens dafür entwickelten `Parser`. Diese Flexibilität setzt aber eine genau vorgegebene **Struktur der Delta-Datei** voraus: Neben den eigentlichen Datenveränderungen der Quellen und den historisierungsspezifischen Angaben wie Gültigkeitszeit und Transaktionszeit muss die Delta-Datei weitere Steuerungsinformationen enthalten. Wenn man davon ausgeht, dass ungeachtet der Historisierungsart und des Datenmodells jede Historisierung durch Aufruf der Methoden `insert()`, `update()` und `delete()` erbracht werden kann, kann die Struktur der Delta-Datei unabhängig von der Historisierungsart und vom zugrunde liegenden Datenmodell definiert werden: Neben den eigentlichen Daten benötigt man lediglich die zusätzliche Information, um welche Datenbankoperation es sich handelt, auf welche Datenbankeinheit sich diese bezieht und welche Historisierungskomponente die Operation ausführen soll. Die Delta-Datei verwendet die folgende, auf unterschiedlichen Ebenen basierte Struktur:

- ❖ **1. Ebene:** Angabe der zu verwendenden Historisierungskomponente. Im Prototyp ist dies aufgrund der erwähnten Einschränkung `HistCompRDB_1`.
- ❖ **2. Ebene:** Angabe der Datenbankeinheit, auf welche die Datenbankoperationen anzuwenden sind. Im Prototyp ist dies aufgrund der Einschränkung auf das relationale Datenmodell ein Relationsname.
- ❖ **3. Ebene:** Angabe der von der jeweiligen Historisierungskomponente durchzuführenden Datenbankoperation, d.h. `insert()`, `update()` oder `delete()`.
- ❖ **4. Ebene:** Parameter der Datenbankoperationen `insert()`, `update()` oder `delete()`. Dabei kann es sich um eigentliche Werte oder wiederum um Datenbankeinheiten handeln, wie beispielsweise Attributnamen im relationalen Datenmodell.
- ❖ **5. Ebene:** Wird verwendet, falls ein Parameter einer Datenbankoperation selbst aus mehreren Elementen besteht.

Um die soeben beschriebene Struktur umzusetzen, benötigt man **drei Trennwörter**. Im Prototyp sind das die folgenden drei:

~<~ Bezeichnet den Beginn einer neuen Ebene.

~>~ Schliesst die aktuelle Ebene.

~;~ Trennt einzelne Elemente gleicher Stufe voneinander ohne eine neue Ebene zu öffnen bzw. eine alte zu schliessen.

Abschliessend sollen nun die in Abbildung 5-1 wiederum als zusammengehörig skizzierten Klassen **HistoryComponent**, **HistCompRDB** und **HistCompRDB_1** genauer betrachtet werden. Wie bereits erwähnt, werden die Methoden `insert()`, `update()` und `delete()` in der Methode `parseDeltaFile()` der Klasse `Parser` aufgerufen. Dazu instanziiert `Parser` zur Laufzeit die jeweilige, in der Delta-Datei angegebene Historisierungskomponente. Die Methode `setEntity()` in der Klasse `HistoryComponent`

erlaubt schliesslich die ebenfalls in der Delta-Datei angegebene Datenbankeinheit entsprechend festzulegen. Beim Prototyp handelt es sich dabei aufgrund der Einschränkung auf das relationale Datenmodell um einen Relationennamen. Deshalb deklariert auch die Unterklasse HistCompRDB die geschützte Klassenvariable `relName`, welcher mittels `setEntity()` der entsprechende Wert zugewiesen wird. Als weitere Klassenvariablen führt HistCompRDB `now` und `uc` auf. Dabei handelt es sich um die bereits in Abschnitt 3.2.1 diskutierten Werte für Gültigkeitszeit und Transaktionszeit. Für die Implementierung wurde für `now` der in Oracle maximale Datumswert 31.12.4712 und für `uc` der Zeitstempel 01.01.0001 00:00:00 gewählt. Es ist zudem festzuhalten, dass der Prototyp auch direkt die Verwendung der Wörter „now“ und „uc“ in der Delta-Datei erlaubt und durch den entsprechenden numerischen Wert ersetzt. Überhaupt verlangt der Prototyp, dass Zeitangaben in der Delta-Datei eine bestimmte Formatierung aufweisen. Um jene dann auf die Java-internen Datum-Klassen `Date`, `Time` und `Timestamp` abzubilden, deklariert HistCompRDB die Klassenmethoden `convertToDateFormat()`, `convertToTimeFormat()` und `convertToTimestampFormat()`. Die Klassenmethoden `getColumnValue()` und `updateColumnValue()` in HistCompRDB liefern den Spaltenwert eines selektierten Tupels zurück bzw. setzen diesen unter Berücksichtigung des jeweiligen Datentyps. Dabei greifen diese Methoden zur Realisierung ihrer Aufgaben auf die Metadaten des Data Warehouse zu. Auch die Methode `getAllColumns()`, welche die Namen aller Attribute einer Relation auflistet, verwendet Data-Warehouse-Metadaten. Die Klasse HistCompRDB_1 schliesslich implementiert als effektive Historisierungskomponente die Methoden `insert()`, `update()` und `delete()`. Zudem deklariert sie als Singleton die Klassenmethode `getInstance()`.

5.1.3 Testumgebung

Wie bereits eingangs erwähnt, umfasst die Testumgebung eine SQL-Start-Datei und eine Delta-Datei. Da zum Testen des Prototyps die schon aus Abschnitt 3.2.1 bekannte Relation KUNDE verwendet werden soll, muss die **SQL-Start-Datei** die Relation KUNDE erstellen und diese dann mit den Ausgangstupeln bevölkern. Nach Ausführen der Start-Datei erhält man das in Tabelle 5-1 dargestellte initiale Data Warehouse, welches nach vollbrachter Historisierung mit Tabelle 3-6 übereinstimmen sollte.

KUNDE

KdNr	NName	VName	Strasse	PLZ	GebDatum	GültigAb	GültigBis	Eingesetzt	Ersetzt
1	Xiang	Lian	Lotosblütenweg 7	8002	13.11.41	01.01.90	now	04.01.1990, 10:14:33	uc
3	Tell	Köbi	Käshaldenstrasse 3	8052	01.08.35	01.01.60	now	10.01.1960, 16:54:01	uc
4	Kurz	Hilde	Hopfenweg 34	3011	22.06.52	01.05.91	now	30.04.1991, 17:33:04	uc

Tabelle 5-1: Initiales Data Warehouse des Prototyps

Die **Delta-Datei** hat schliesslich im Fall des Prototyps das in Abbildung 5-2 gezeigte Aussehen. Es ist darauf hinzuweisen, dass der Prototyp die Erstellung der Delta-Datei nicht behandelt, sondern als gegeben vorausgesetzt. Diese Erfordernis basiert auf der Annahme, dass die in Abschnitt 2.2.1 erläuterten Akquisitionsschritte sequentiell durchlaufen werden und dass deshalb die Delta-Datei bereits in einem vorangehenden Akquisitionsschritt durch eine eigens dafür entwickelte Komponente erzeugt worden ist. Dem ist aber in der Praxis meist nicht so: Erfahrungsgemäss werden die konzeptionell unterschiedlichen Akquisitionsschritte nicht sauber voneinander getrennt, sondern teilweise miteinander kombiniert. Insofern würde in der Praxis höchstwahrscheinlich auch die Erstellung der Delta-Datei in die Kompetenz des Historisierungswerkzeugs fallen.

```

~<~HistCompRDB_1~<~CUSTOMER~<~Update~<~Nr~;~3~<~Street~;~ZIP~>~<~Läckerligasse 1~;~4051~>~
19740930~;~19741001~;~19741013075513~>~Update~<~Nr~;~3~<~Street~;~ZIP~>~<~Käshaldenstrasse 3~;~
8052~>~19861231~;~19870101~;~19870107150200~>~Delete~<~Nr~;~4~;~19940930~;~19941030080402~>~
Update~<~Nr~;~1~<~Street~;~ZIP~>~<~Drachenstrasse 13~;~8004~>~19991231~;~20000101~;~
20000105143527~>~Insert~<~<~Nr~;~LName~;~FName~;~Street~;~ZIP~;~BDate~>~<~2~;~Eiffel~;~Pierre~
;~Am Triumphbogen~;~8200~;~19630714~>~20001001~;~20001001093108~>~>~>~>~

```

Abbildung 5-2: Delta-Datei des Prototyps

5.2 ERKENNTNISSE

Der in den vorangehenden Abschnitten erläuterte Prototyp scheint der eingangs an ihn gestellten Anforderung gerecht zu werden. Dadurch, dass er auf dem in Abschnitt 4.2 erarbeiteten Metadatenschema basiert, wahrt er eine gewisse Offenheit gegenüber Historisierungsart und Datenmodell: Durch Implementierung neuer Historisierungskomponenten lassen sich weitere datenmodellspezifische Historisierungsarten realisieren und ohne Änderungen an den bestehenden Programmklassen in den Prototyp integrieren. Weiter liegt eine Metadatensteuerung vor. Auf den ersten Blick scheint jedoch die Historisierungskomponente des Prototyps weniger Metadaten zu verwenden, als zuerst erwartet. Es lässt sich somit fragen, ob es andere Implementierungsmöglichkeiten für den Prototyp gibt, die zu einer extensiveren Metadatensteuerung führen und insgesamt der soeben vorgestellten Lösung vorzuziehen sind. Im folgenden werden zwei Möglichkeiten diskutiert:

String-Datentypkonversion während dem Einlesen der Delta-Datei. Wie bereits in Abschnitt 5.1.2 erwähnt, verwenden lediglich die Klassenmethoden `getColumnValue()` und `updateColumnValue()` und die Instanzmethode `getAllColumns()` Metadaten. Dabei geht es um Metadaten über den Aufbau und die Struktur des Data Warehouse. Für die weiteren Betrachtungen von Interesse sind aber nur noch die Klassenmethoden `getColumnValue()` und `updateColumnValue()`. Während die Methode `getColumnValue()` zum Kopieren eines Tupels verwendet wird und dazu gelesene Spaltenwerte in Objekte umwandelt, erlaubt die Methode `updateColumnValue()` den Spaltenwert mit dem als Parameter übergebenen String-Wert zu überschreiben. Beide Methoden benötigen zur Datentypkonversion die Information, um welchen Datentyp es sich in der jeweiligen Spalte handelt. Es zeichnet sich nun die Möglichkeit ab, die Implementierung dahingehend zu verändern, dass die Konversion eines String-Wertes in den jeweiligen Spaltentyp nicht erst beim Einfügen oder Ändern eines Tupels durch die Methode `updateColumnValue()` passiert, sondern bereits während dem Einlesen der Delta-Datei geschieht. Die Programmklasse `Parser` müsste also entsprechend geändert werden.

Der folgende Ansatz bietet sich dabei an: Für jede Implementierung der Methoden `insert()`, `update()` und `delete()` – und somit für jede Historisierungskomponente – müssen Metadaten gehalten werden. Diese bestimmen, von welcher Klasse der Java-Klassenbibliothek eine Instanz zu erzeugen ist, um dann als Parameter der jeweiligen Modifikationsoperation übergeben zu werden. Etwaig benötigte Werte für die Erzeugung einer Instanz werden dabei aus der Delta-Datei gelesen. Zur Verdeutlichung sei das Ganze an der Historisierungskomponente `HistCompRDB_1` illustriert. Tabelle 5-2 zeigt die zu verwendenden Metadaten, Abbildung 5-3 die zur metadatengesteuerten Instanzierung aufzurufenden Methoden. Dabei wurde in Abbildung 5-3 der Übersichtlichkeit wegen absichtlich auf die Angabe der Exceptions im Methodenkopf verzichtet.

MetaHistCompRDB_1

Operation	Entry	Name	ClassName	MethodName	ParameterTypes
insert	1	names	java.util.Vector	null	null
insert	2	values	java.util.Vector	null	null
insert	3	VST	java.sql.Date	valueOf	java.lang.String
insert	4	TST	java.sql.Timestamp	valueOf	java.lang.String
insert	5	VET	java.sql.Date	valueOf	java.lang.String
update	1	ntPKeyName	java.lang.String	null	java.lang.String
update	2	ntPKeyValue			
update	3	names	java.util.Vector	null	null
update	4	values	java.util.Vector	null	null
update	5	oldVET	java.sql.Date	valueOf	java.lang.String
update	6	newVST	java.sql.Date	valueOf	java.lang.String
update	7	newTT	java.sql.Timestamp	valueOf	java.lang.String
update	8	newVET	java.sql.Date	valueOf	java.lang.String
delete	1	ntPKeyName	java.lang.String	null	java.lang.String
delete	2	ntPKeyValue			
delete	3	VET	java.sql.Date	valueOf	java.lang.String
delete	4	TT	java.sql.Timestamp	valueOf	java.lang.String

Tabelle 5-2: Metadaten für die Historisierungskomponente HistCompRDB_1

Tabelle 5-2 weist für jede der Modifikationsoperationen insert(), update() und delete() mehrere Tupel auf, nämlich für jeden Parameter eines. Die ersten beiden Spalten werden für diese Angaben verwendet. Die dritte Spalte führt den Namen des Parameters auf. Dieser wird jedoch nicht für die Metadatensteuerung verwendet, sondern soll allein dem Benutzer die Lesbarkeit und Interpretation der Zeilen erleichtern. Die Spalte ClassName bestimmt schliesslich jene Klasse der Java-Klassenbibliothek, welche instanziiert werden soll. Dabei ist diejenige Java-Klasse zu verwenden, die dem jeweiligen Datenbank-Datentyp der Spalte entspricht. Für die Instanzierung kommen Konstruktoren und Klassenmethoden in Frage. Falls ein Konstruktor zum Einsatz gelangen soll, weist die Spalte MethodName den Wert null auf. Ansonsten gibt die Spalte MethodName den Namen der aufzurufenden Klassenmethode an. Der Datentyp der für die Instanzierung zu verwendenden Parameter – ob Konstruktor oder Klassenmethode – ist der Spalte ParameterTypes zu entnehmen. Während der Datentyp dieser Parameter in der Tabelle 5-2 festgehalten ist, müssen deren Werte aus der Delta-Datei eingelesen werden. Sollte ein Konstruktor bzw. eine Klassenmethode mehrere Parameter deklarieren, ist dabei für jeden dieser Parameter ein separater Eintrag zu setzen. Aus diesem Grund ist auch der Primärschlüssel von MetaHistCompRDB_1 eine Kombination der Spalten Operation, Entry und ParameterTypes. Falls ein Konstruktor bzw. eine Klassenmethode aber keine Parameter deklariert, ist der Eintrag in der Spalte ParameterTypes konsequenterweise null. Es ist noch darauf hinzuweisen, dass die in Tabelle 5-2 aufgeführten Klassennamen voll qualifiziert sein müssen, was bedeutet, dass auch die Angabe des Package erfolgen muss.

Die Metadaten aus Tabelle 5-2 werden nun in den Methoden create() in Abbildung 5-3 verwendet. Analog zur Unterscheidung in der Tabelle gibt es auch in der Abbildung zwei Methoden, je eine für die Instanzierung durch einen Konstruktor bzw. durch eine Klassenmethode: Die in der Abbildung 5-3

zuerst aufgeführte Methode `create()` verwendet einen Konstruktor, die als Zweites erwähnte Methode eine Klassenmethode. Grundsätzlich machen die Methoden `create()` Gebrauch von der Java-Klasse `Class` und der in ihr definierten Methoden `forName()`, `getDeclaredConstructor()` und `getDeclaredMethod()`. Diese Methoden erlauben die Erzeugung von Klasseninstanzen, Konstruktoren oder Methoden aufgrund der Angabe ihres Namens. Sie schlagen somit gewissermassen die Brücke zwischen einem String-Wert und dem durch ihn bezeichneten Objekt.

Zur Illustration sei die Instanziierung des dritten Parameters der Methode `insert()` in Tabelle 5-2 erläutert. Dabei handelt es sich um die Angabe des Startzeitpunktes in Gültigkeitszeit (GültigAb, engl. VST). Da im Data Warehouse für die Gültigkeitszeit der SQL-Datentyp `Date` verwendet wird, kommen also grundsätzlich die Java-Klassen `Date`, `Time` und `Timestamp` in Frage. Aufgrund ihrer Granularitätsstufe kommt schliesslich die Java-Klasse `Date` für die Gültigkeitszeit zum Zug, was aus der vierten Spalte in Tabelle 5-2 hervorgeht. Die fünfte Spalte zeigt an, dass zur Instanziierung nicht ein Konstruktor, sondern die Klassenmethode `valueOf()` verwendet werden soll. Die Parameter bzw. deren Typ finden sich in der letzten Spalte. Es gelangt somit die zweite Methode in Abbildung 5-3 zum Einsatz. Dabei werden die ersten drei Parameter mit den Metadaten „`java.sql.Date`“, „`valueOf`“ und „`java.lang.String`“ aus der Tabelle `MetaHistCompRDB_1` gespiesen, der letzte Parameter `values` hingegen, muss aus der Delta-Datei eingelesen werden.

```
public static Object create(String className, String[] parameterTypes,
                           Object[] values) {
    Class c=Class.forName(className);
    if (parameterTypes != null) {
        int parameterNumber=parameterTypes.length;
        Class[] parameter=new Class[parameterNumber];
        for (int i=0; i<parameterNumber; i++) {
            parameter[i]=Class.forName(parameterTypes[i]);
        }
        Constructor con=c.getDeclaredConstructor(parameter);
        return con.newInstance(values);
    }
    else {
        Constructor con=c.getDeclaredConstructor(null);
        return con.newInstance(null);
    }
}

public static Object create(String className, String methodName,
                           String[] parameterTypes, Object[] values) {
    Class c=Class.forName(className);
    if (parameterTypes != null) {
        int parameterNumber=parameterTypes.length;
        Class[] parameter=new Class[parameterNumber];
        for (int i=0; i<parameterNumber; i++) {
            parameter[i]=Class.forName(parameterTypes[i]);
        }
        Method m=c.getDeclaredMethod(methodName, parameter);
        return m.invoke(null, values);
    }
    else {
        Method m=c.getDeclaredMethod(methodName, null);
        return m.invoke(null, null);
    }
}
```

Abbildung 5-3: Methoden zur dynamischen, metadaten gesteuerten Klassen-Instanziierung

Die bisherigen Ausführungen scheinen nahe zu legen, den soeben diskutierten Ansatz als potentiellen Konkurrenten zu wählen. Führt man sich jedoch die folgenden Punkte vor Augen, ist der Ansatz weit nicht mehr so attraktiv: Bisweilen wurden die leeren Zellen in Tabelle 5-2 unterschlagen. Genau dort aber lässt sich die Schwachstelle dieses Ansatzes finden. Weil der Datentyp des nicht-temporalen Primärschlüssels für jede Relation spezifisch ist und somit nicht für eine Historisierungskomponente festgelegt werden kann, ist man gezwungen, diesen zur Laufzeit durch Verwendung des Attributnamens `ntPKeyName` zu ermitteln. Dazu benötigt man – wie auch im Prototyp – Angaben über den Aufbau und die Struktur des Data Warehouse. Dieselben Überlegungen gelten auch für die zu ändernden Attribute. Zwar kann man den Vector values mit Hilfe der Metadaten aus Tabelle 5-2 instanzieren, für das Abfüllen des Vectors mit den aus der Delta-Datei gelesenen Werten benötigt man aber für jeden Wert die Information, welcher Datentyp zu verwenden ist. Es zeigt sich somit, dass dieser Ansatz zusätzlich zu den Metadaten in Tabelle 5-2 auch die bereits im Prototyp eingesetzten Data-Warehouse-Metadaten benötigt. Zudem darf nicht vergessen werden, dass der vorliegende Ansatz lediglich die Auslagerung der String-Konvertierung realisiert, die im Prototyp durch die Methode `updateColumnValue()` erbracht wird. Die Methode `getColumnValue()` hingegen wird weiterhin zum Kopieren von Tupeln innerhalb der Methoden `update()` und `delete()` verwendet, wozu sie Angaben über den Datentyp einer Spalte benötigt. Die bisherigen Ausführungen machen also deutlich, dass der vorliegende Ansatz wohl zu einer extensiveren Metadatensteuerung führen würde, man aber dadurch keinerlei Vorteile gewinnt – die Funktionalität bleibt dieselbe. Im Gegenteil: In Anbetracht des enormen Aufwands, der für die Verwaltung von Metadaten anfällt, ist der hinsichtlich des Metadateneinsatzes „leichtere“ Prototyp diesem „aufgeblähten“ Ansatz vorzuziehen.

Keine Implementierung der Historisierungskomponenten. Als weitere Möglichkeit zeichnet sich der Ansatz ab, die Historisierungskomponenten in Abbildung 4-1 nicht auf die Java-Applikation abzubilden. In diesem Fall würden lediglich die datenmodellspezifischen Oberklassen `HistCompOODB` und `HistCompRDB` implementiert, die Historisierungskomponenten aber nur als Metadaten gehalten. Die Programmklassen `HistCompOODB` und `HistCompRDB` müssten dann also so allgemein implementiert werden, dass sie zwar datenmodellspezifisch sind, aber gegenüber jeglicher Historisierungsart offen bleiben. Erst durch Einbindung der Metadaten wird ein historisierungsspezifisches Verhalten realisiert. Weil aber die Implementierung der Methoden `insert()`, `update()` und `delete()` für jeden Punkt im Historisierungsraum unterschiedlich ausfällt, müssten diese Methoden ebenfalls als Metadaten – als so genannte Stored Procedures – ausgelagert werden. Die Komplexität bei diesem Ansatz verlagert sich im Vergleich zum Prototyp von der Applikation auf das Repository. Ansonsten sind sich beide Ansätze ebenbürtig. Wie gesagt, bedingt aber der nun diskutierte Ansatz den Einsatz von Stored Procedures. Da solche Stored Procedures aber in einem rein relationalen Datenbanksystem nicht zur Verfügung stehen, verbleibt für den rein relationalen Fall lediglich der Prototyp zur Auswahl.

6 ZUSAMMENFASSUNG

Zu Beginn dieser Arbeit musste ich feststellen, dass es bis zum heutigen Zeitpunkt in der Literatur noch keine einheitliche Betrachtungsweise für die Historisierung gibt. Ich stiess lediglich auf vereinzelte Historisierungstechniken, die aus verschiedenen Teilgebieten der Datenbanktechnologie hervorgingen und deshalb einen starken gebietsspezifischen Charakter aufweisen. Zum einen waren dies Konzepte aus den temporalen Datenbanken, zum anderen Konzepte aus der multidimensionalen Datenmodellierung. Ich hielt also gewissermassen zwei Mosaiksteinchen eines mir noch unbekannten grossen Mosaiks in der Hand und fragte mich, ob ich durch eingehende Auseinandersetzung mit dem mir Gegebenen die Gemeinsamkeiten herausfiltern und dadurch eine einheitliche Betrachtungsweise gewinnen könnte.

Es war also mein Ziel, ein Rahmenmodell für die Historisierung zu schaffen, welches zum einen die bereits in der Literatur und Praxis bestehenden Historisierungskonzepte unter einem Blickwinkel vereint und zum anderen eine gewisse Offenheit wahrt, um auch künftig entdeckte Historisierungstechniken einzubetten. Schon bald einmal bot sich die Verwendung von Historisierungsdimensionen aufgrund ihrer Orthogonalität als vielversprechender Modellierungsansatz an, weshalb ich mich fortan darauf konzentrierte, möglichst alle Dimensionen eines solchen abstrakten Historisierungsmodells zu finden.

Aus der Auseinandersetzung mit den temporalen Datenbanken konnte ich die folgenden vier Dimensionen des abstrakten Historisierungsmodells gewinnen: Zeitdimensionen, Zeitgranulat, Entwicklungsprozess und Datengranulat. In den temporalen Datenbanken wird die Verwendung von unterschiedlichen Zeitdimensionen [Ste] und der konkrete Einsatz von Gültigkeits- und Transaktionszeit [ElNa] ausführlich diskutiert. Zudem wird auch erwähnt, dass Gültigkeits- und Transaktionszeit meist in unterschiedlicher Genauigkeit angegeben werden. Auch die Tupel- und Attribut-Versionierung werden als unterschiedliche Konzepte einander gegenübergestellt, wobei [Ste] bereits darauf hinweist, dass sich die Versionierung auch auf andere Einheiten, wie beispielsweise die Extension, ausdehnen kann. Aufgrund dieser durch [ElNa] und [Ste] erbrachten Vorarbeit, fiel die Wahl und Ausgestaltung der Dimensionen Zeitdimensionen, Zeitgranulat und Datengranulat leicht aus.

Die Dimension Entwicklungsprozess verlangte jedoch mehr Ausarbeitung. Als Ausgangslage diente mir die Bemerkung in [Ste] und [ElNa], dass in temporalen Datenbanken sowohl Zeitpunkte als auch Zeitintervalle zum Einsatz gelangen. Der in [ElNa] gemachte Erklärungsversuch, dass Zeitpunkte zur Beschreibung von „Ereignis-Information“ (engl. event information) und Zeitintervalle zur Beschreibung von „Zustands-Information“ (engl. duration information, state information) verwendet werden, schien mich insofern nicht zu befriedigen, dass ich dadurch noch immer keine konkrete Handhabung hatte, wann welches Konzept zu gebrauchen ist. Erst als ich in [Schä] auf die unterschiedlichen Entwicklungsprozess-Typen einer Entität der Realwelt stiess, kam mir die Idee, die Wahl des Modellierungskonzeptes – Zeitpunkt oder Zeitintervall bzw. (in der in dieser Arbeit verwendeten Terminologie) Momentaufnahme oder Zustandsdauer – vom zugrunde liegenden Entwicklungsprozess abhängig zu machen. Aber auch die in [Schä] vorgestellten vier Entwicklungsprozess-Typen schienen mir in irgendeiner Weise widersprüchlich zu sein, zumal sich für mich der stufenweise konstante Entwicklungsprozess nicht wesentlich vom diskreten zu unterscheiden schien: Ich hatte das Gefühl, dass es sich um denselben Entwicklungsprozess, aber um unterschiedliche Sichtweisen darauf handle. Diese Einsicht brachte mich zur Erkenntnis, dass es zwei Möglichkeiten geben muss, einen stufenweise konstanten Entwicklungsprozess zu modellieren: Eine zustandsorientierte und eine ereignisorientierte. Als ich mich später im Verlauf der Arbeit mit dem temporalen Star-Schema beschäftigte, stiess ich in [BJSS] zufälligerweise auf dieselben Überlegungen. Insofern fühle ich mich in der Richtigkeit meiner Darlegungen in Abschnitt 3.1.3 bestätigt.

Dank der Auseinandersetzung mit der multidimensionalen Datenmodellierung gelangte ich schliesslich zu den restlichen Dimensionen des abstrakten Historisierungsmodells: Zeitangabe, Entwicklungsbeschreibung und Wachstumsrichtung. Die Tatsache, dass im herkömmlichen Star-Schema der Zeitbezug über eine Primär-Fremdschlüssel-Beziehung hergestellt wird, verdeutlichte mir, dass es neben einer expliziten Zeitangabe auch die Möglichkeit zu einer impliziten Zeitangabe gibt. Weiter machte mich das Verfahren nach Typ 3 für den Umgang mit Slowly Changing Dimensions darauf aufmerksam, dass die Entwicklungsbeschreibung auch lückenhaft ausfallen und dass es auch eine horizontale Wachstumsrichtung geben kann.

Nach Erarbeitung des abstrakten Historisierungsmodells war der Weg für die Ausgestaltung des Repository-Schemas und die Implementierung eines Historisierungswerkzeugs geebnet: Durch Einbezug des abstrakten Historisierungsmodells und dem Zusammentragen sämtlicher historisierungsrelevanter Informationen gelangte ich zu einem Repository-Schema, welches mir in sich konsistent erschien. Die Implementierung fiel dadurch leicht aus, dass ich die Programmarchitektur vollumfänglich am Metadaten-Klassendiagramm ausrichten konnte und diese somit gewissermassen gegeben war. Nebst den üblichen Programmierfallen war für mich die Implementation eines hinsichtlich Historisierungsart und Datenmodell unabhängigen Parsers eine interessante Herausforderung.

Diese Arbeit hat mir trotz aller begleitenden „Hochs und Tiefs“ sehr viel Freude bereitet. Falls ich damit die eingangs erwähnte Lücke in der Datenbankliteratur ein wenig füllen und dadurch vielleicht eine gute Ausgangslage für ein anderes Projekt schaffen konnte, wäre dies ein riesiges Kompliment für mich.

ANHANG A: QUELLTEXT ZUM PROTOTYP

```

/*****
/* Author:      Stephanie Balzer
/* Project:     Development of a metadata driven tool to build the history of
/*              data in data warehouses
/* Copyright:   Department of Information Technology
/*              Database Technology Research Group
/*              University of Zurich, Switzerland
/* Language:    Java, Sun's Java Development Kit, Version 1.3.1
/* Class:       Run
/* Version:     1.0
/* Date:        17.09.2001
/*-----
/* Description:
/*
/* Class Run is the starting point of the application and therefore contains
/* method main(). Main requires the following arguments, which have to be
/* supplied by the user: name of the delta file, user's service name (entry in
/* tnsname.ora), login and password. Optionally the character encoding used for
/* the delta file can be indicated as fifth parameter. The read arguments are
/* passed to the class Manager.
*****/

```

```

import java.io.*;
import java.sql.*;

public class Run {

    public static void main(String[] args) {
        try {
            if ((args.length < 4) || (args.length > 5)) {
                throw new NumberFormatException("The main method requires as " +
                    "input parameters the name of the delta file, the user's " +
                    "service name (entry in tnsname.ora), login and password. " +
                    "Optionally as fifth parameter the name of the character " +
                    "encoding can be indicated");
            }
            else {
                String deltaFileName=args[0];
                String serviceName=args[1];
                String login=args[2];
                String password=args[3];
                Manager mngr;
                if (args.length==4) {
                    mngr=new Manager(deltaFileName, serviceName, login, password);
                }
                else {
                    String encoding=args[4];
                    mngr=new Manager(deltaFileName, serviceName, login, password,
                        encoding);
                }
                mngr.buildHistory();
                System.out.println("Updates to the database have been " +
                    "successfully applied.");
            }
        }
        //Catches exceptions that were thrown before any updates were made
        //to the database. Therefore no rollback is needed.
        catch (NumberFormatException e) {
            e.printStackTrace(System.out);
        }
    }
}

```



```
        catch (FileNotFoundException e) {  
            e.printStackTrace(System.out);  
        }  
        catch (IOException e) {  
            e.printStackTrace(System.out);  
        }  
    }  
}
```

```

/*****
/* Author:      Stephanie Balzer
/* Project:     Development of a metadata driven tool to build the history of
/*              data in data warehouses
/* Copyright:   Department of Information Technology
/*              Database Technology Research Group
/*              University of Zurich, Switzerland
/* Language:    Java, Sun's Java Development Kit, Version 1.3.1
/* Class:       Manager
/* Version:     1.1
/* Date:        15.10.2001
/* -----
/* Description:
/*
/* The arguments read in the method main() of class Run are passed to the class
/* Manager, which stores them as instance variables. The Manager provides
/* methods to open and close a Connection to the target database and creates a
/* Statement for database manipulation. AutoCommit is disabled to encapsulate
/* all the updates made to the database in a single transaction that can be
/* committed as a whole or in case of failure rolled back as a whole. At the
/* heart of Manager is the method buildHistory() which basically invokes method
/* parseDeltaFile() of class Parser.
*****/

import java.lang.reflect.*;
import java.io.*;
import java.util.*;
import java.sql.*;

public class Manager {
    //Information about the delta file and the target database.
    private String deltaFileName;
    private String serviceName;
    private String login;
    private String password;
    private String encoding;

    //Connection to the target database and
    //statement for target database manipulation.
    private Connection con;
    private Statement stmt;

    //The instance parser parses the delta file and invokes the appropriate
    //method insert(), update() or delete().
    private Parser parser;

    public Manager(String deltaFileName, String serviceName, String login,
                   String password) throws FileNotFoundException, IOException {
        this.deltaFileName=deltaFileName;
        this.serviceName=serviceName;
        this.login=login;
        this.password=password;
        this.parser=new Parser(deltaFileName);
    }

    //Constructor that additionally allows specifying the character encoding
    //for parsing the delta file.
    public Manager(String deltaFileName, String serviceName, String login,
                   String password, String encoding) throws FileNotFoundException,
                                                           IOException {
        this.deltaFileName=deltaFileName;
        this.serviceName=serviceName;
        this.login=login;
        this.password=password;

```

```
        this.encoding=encoding;
        this.parser=new Parser(deltaFileName, encoding);
    }

    //Tries to connect to the target database and returns a Statement for
    //database manipulation. AutoCommit is disabled to encapsulate all the
    //updates made to the database in a single transaction that can be committed
    //as a whole or in case of failure rolled back as a whole.
    public void connectToDatabase() throws ClassNotFoundException, SQLException {
        Class.forName("oracle.jdbc.driver.OracleDriver");
        con=DriverManager.getConnection("jdbc:oracle:oci8:@" + serviceName,
                                         login, password);

        con.setAutoCommit(false);
        stmt=con.createStatement(ResultSet.TYPE_SCROLL_INSENSITIVE,
                                  ResultSet.CONCUR_UPDATABLE);
    }

    public void disconnectFromDatabase() throws SQLException {
        stmt.close();
        con.close();
    }

    public void buildHistory() {
        try {
            connectToDatabase();
        }
        //Catches exceptions that were thrown during connection phase.
        //Therefore no rollback is executed.
        catch (ClassNotFoundException e) {
            e.printStackTrace(System.out);
        }
        catch (SQLException e) {
            e.printStackTrace(System.out);
            System.out.println("SQLstate: " + e.getSQLState());
            System.out.println("error code: " + e.getErrorCode());
        }
        try {
            parser.parseDeltaFile(stmt);
            con.commit();
        }
        //Catches exceptions that were thrown while the database was being updated.
        //Therefore the database is rolled back.
        catch (SQLException e) {
            e.printStackTrace(System.out);
            System.out.println("SQLstate: " + e.getSQLState());
            System.out.println("error code: " + e.getErrorCode());
            try {
                System.out.println("Transaction is being rolled back");
                con.rollback();
            }
            catch (SQLException ex) {
                e.printStackTrace(System.out);
                System.out.println("SQLstate: " + ex.getSQLState());
                System.out.println("error code: " + ex.getErrorCode());
            }
        }
        catch (Exception e) {
            e.printStackTrace(System.out);
            try {
                System.out.println("Transaction is being rolled back");
                con.rollback();
            }
            catch (SQLException ex) {
                e.printStackTrace(System.out);
                System.out.println("SQLstate: " + ex.getSQLState());
            }
        }
    }
}
```

```
        System.out.println("error code: " + ex.getErrorCode());
    }
}
finally {
    try {
        disconnectFromDatabase();
    }
    catch (SQLException e) {
        e.printStackTrace(System.out);
        System.out.println("SQLstate: " + e.getSQLState());
        System.out.println("error code: " + e.getErrorCode());
    }
}
}
```

```

/*****
/* Author:      Stephanie Balzer
/* Project:     Development of a metadata driven tool to build the history of
/*              data in data warehouses
/* Copyright:   Department of Information Technology
/*              Database Technology Research Group
/*              University of Zurich, Switzerland
/* Language:    Java, Sun's Java Development Kit, Version 1.3.1
/* Class:       Parser
/* Version:     1.2
/* Date:        18.10.2001
/* -----
/* Description:
/*
/* Reads in a delta file containing historical information which has to be
/* transferred accordingly to the target database. The delta file has a five
/* level architecture. The first level specifies which HistoryComponent to use
/* for the updates to the target database. The second denominates the database
/* entity to which the updates are applied. The third level indicates the kind
/* of database operation (insert, update or delete) which has to be executed.
/* The following two levels finally provide the actual values. The fifth level
/* thereby allows grouping several related values together. To parse the delta
/* file the Parser uses an instance of class InputFileTokenizer. The parsing is
/* done in method parseDeltaFile(), which basically invokes the appropriate
/* method (insert, update or delete) of the HistoryComponent to use.
*****/

import java.lang.reflect.*;
import java.io.*;
import java.util.*;
import java.sql.*;

public class Parser {
    private InputFileTokenizer ift;
    private HistoryComponent hc;
    private String operation;

    //Parameters that are passed to the respective database operation.
    private Vector parameters;
    private Vector vectorParameter;

    //Three delimiters are used. One to indicate the beginning of a new level
    //(openLevel) and one to indicate the end of the actual level (closeLevel).
    //The third delimiter is used to separate values (separate).
    private final String openLevel="\u007e\u003c\u007e";
    private final String closeLevel="\u007e\u003e\u007e";
    private final String separate="\u007e\u003b\u007e";

    //Indicates the actual level.
    private int level;

    public Parser(String deltaFileName) throws FileNotFoundException, IOException {
        ift=new InputFileTokenizer(deltaFileName);
        ift.setDelimiter(openLevel);
        ift.setDelimiter(closeLevel);
        ift.setDelimiter(separate);
    }

    //Constructor that additionally allows specifying the character encoding
    //for parsing the delta file.
    public Parser(String deltaFileName, String encoding)
        throws FileNotFoundException, IOException {
        ift=new InputFileTokenizer(deltaFileName, encoding);
        ift.setDelimiter(openLevel);
        ift.setDelimiter(closeLevel);
    }

```

```

        ift.setDelimiter(separate);
    }

    public void execute() throws SQLException, TemporalFormatException,
        NumberOfArgumentException {
        if (operation.equalsIgnoreCase("Insert")) {
            hc.insert(parameters);
        }
        else {
            if (operation.equalsIgnoreCase("Update")) {
                hc.update(parameters);
            }
            else {
                hc.delete(parameters);
            }
        }
    }
}

public void openLevel(Statement stmt) throws DeltaFileException {
    level++;
    switch (level) {
        case 1: break;
        case 2: break;
        case 3: parameters=new Vector();
                parameters.add(stmt);
                break;
        case 4: break;
        case 5: vectorParameter=new Vector();
                break;
        default: throw new DeltaFileException("The delta file does not have " +
            "the requested structure.");
    }
}

public void closeLevel(Statement stmt) throws SQLException,
    TemporalFormatException,
    DeltaFileException,
    NumberOfArgumentException {

    level--;
    switch (level) {
        case 0: break;
        case 1: break;
        case 2: break;
        case 3: execute();
                parameters=new Vector();
                parameters.add(stmt);
                break;
        case 4: parameters.add(vectorParameter);
                break;
        default: throw new DeltaFileException("The delta file does not have " +
            "the requested structure.");
    }
}

public void parseDeltaFile(Statement stmt) throws ClassNotFoundException,
    NoSuchMethodException, IllegalAccessException,
    InvocationTargetException, IOException,
    DeltaFileException, SQLException,
    TemporalFormatException, NumberOfArgumentException {

    try {
        ift.next();
        if (ift.isToken) {
            throw new DeltaFileException("The delta file is expected to start " +
                "with a delimiter.");
        }
    }
}

```

```
while (!ift.endOfFile) {
    //The do while statement handles delimiters.
    do {
        if (ift.delimiter.equals(openLevel)) {
            openLevel(stmt);
        }
        else if (ift.delimiter.equals(closeLevel)) {
            closeLevel(stmt);
        }
        ift.next();
    } while (!ift.isToken);
    if (ift.endOfFile) {
        break;
    }
    //The switch statement handles tokens and is therefore only entered
    //if a token was detected.
    switch (level) {
        //Specification of HistoryComponent to use.
        case 1: Class c=Class.forName(ift.token);
                Method m=c.getMethod("getInstance", null);
                hc=((HistoryComponent) m.invoke(null, null));
                ift.next();
                break;

        //Specification of Entity to which update operations are applied.
        case 2: hc.setEntity(ift.token);
                ift.next();
                break;

        //Specification of update operation and
        //initialisation of Vector parameters.
        case 3: operation=ift.token;
                ift.next();
                break;

        //String parameters.
        case 4: parameters.add(ift.token);
                ift.next();
                break;

        //String Vector parameters.
        case 5: vectorParameter.add(ift.token);
                ift.next();
                break;

        default: throw new DeltaFileException("The delta file does not " +
                                                "have the requested structure.");
    }
}
if (level!=0) {
    throw new DeltaFileException("There must be as many closing as " +
                                "opening delimiters in the delta file.");
}
}
finally {
    ift.close();
}
}
```

```

/*****
/* Author:      Stephanie Balzer
/* Project:     Development of a metadata driven tool to build the history of
/*              data in data warehouses
/* Copyright:   Department of Information Technology
/*              Database Technology Research Group
/*              University of Zurich, Switzerland
/* Language:    Java, Sun's Java Development Kit, Version 1.3.1
/* Class:       InputFileTokenizer
/* Version:     1.0
/* Date:        08.10.2001
/* -----
/* Description:
/*
/* An InputFileTokenizer allows parsing a file that contains tokens (text)
/* separated by Delimiters. The user has to specify the delimiters with the
/* method setDelimiter(). Thereby a arbitrary number of delimiters can be
/* specified and each delimiter can be of arbitrary length. The method next()
/* achieves the parsing. After each call of next() the boolean variable isToken
/* indicates whether a token or delimiter was detected. The respective values
/* can be read from token or delimiter. The boolean variable endOfFile can be
/* used to control iteration. An InputFileTokenizer does ignore new line
/* characters but considers spaces to be part of tokens respectively delimiters.
/* A close() method is also provided to close the underlying InputStreamReader.
*****/

import java.io.*;
import java.util.*;

public class InputFileTokenizer {
    private InputStreamReader isr;

    //Holds the previously read character.
    private int actualChar;

    //The hashtable contains as values vectors which in turn hold all delimiters
    //starting with the same character. The hash key is the starting character
    //of a delimiter.
    private Hashtable delimiters;

    //A stack which holds marks. A mark object is generated every time the method
    //mark() is called.
    private Stack marks;

    //This boolean variable is updated in the method getToken() and prevents from
    //checking twice whether a delimiter was detected.
    private boolean delimiterDetected;

    //This boolean variable is needed for the case where a file ends with a token
    //which in turn is very similar to a delimiter, i.e. if the delimiter has the
    //token as its beginning substring.
    private boolean tokenDetected;

    public String delimiter;
    public String token;
    public boolean isToken;
    public boolean endOfFile;

    //This InputFileTokenizer uses the default character encoding. Apart from the
    //initialisation of isr this constructor is identical with the one below:
    public InputFileTokenizer(String inputFile) throws FileNotFoundException,
                                                                    IOException {
        isr=new InputStreamReader(new FileInputStream(inputFile));
        //Ignore new line characters.
        do {

```



```

        actualChar=isr.read();
    }
    while ((actualChar==10) || (actualChar==13));
    //Check whether file is empty.
    if (actualChar != -1) {
        endOfFile=false;
    }
    else {
        endOfFile=true;
        throw new IOException("Empty input file.");
    }
    delimiters=new Hashtable();
    marks=new Stack();
    delimiterDetected=false;
    delimiter=new String();
    token=new String();
}

//This InputFileTokenizer uses the specified character encoding. Apart from
//the initialisation of isr this constructor is identical with the one above.
public InputFileTokenizer(String inputFile, String encoding)
    throws FileNotFoundException, IOException {
    isr=new InputStreamReader(new FileInputStream(inputFile), encoding);
    //Ignore new line characters.
    do {
        actualChar=isr.read();
    }
    while ((actualChar==10) || (actualChar==13));
    //Check whether file is empty.
    if (actualChar != -1) {
        endOfFile=false;
    }
    else {
        endOfFile=true;
        throw new IOException("Empty input file.");
    }
    delimiters=new Hashtable();
    marks=new Stack();
    delimiterDetected=false;
    delimiter=new String();
    token=new String();
}

public void setDelimiter(String DelimiterString) {
    Object o;
    if ((o=delimiters.get(new Character(DelimiterString.charAt(0)))) == null) {
        Vector v=new Vector();
        v.add(new Delimiter(DelimiterString));
        delimiters.put(new Character(DelimiterString.charAt(0)), v);
    }
    else {
        Vector v=(Vector) o;
        v.add(new Delimiter(DelimiterString));
    }
}

public void next() throws IOException {
    if ((delimiterDetected==true) || (isDelimiter())) {
        isToken=false;
        token="";
        delimiterDetected=false;
    }
    else {
        //Rare case where file ends with a token that is the beginning
        //of a delimiter.

```

```

        if (tokenDetected) {
            isToken=true;
            delimiter="";
            tokenDetected=false;
        }
        else {
            isToken=true;
            delimiter="";
            getToken();
        }
    }
}

public void close() throws IOException {
    isr.close();
}

//This method guarantees in case method mark() is called that the read
//characters are written into the mark's queue. This is exactly from where
//they can be reread after a reset.
private void read() throws IOException {
    try {
        Object o=marks.peek();
        Mark m=(Mark) o;
        //Every time isMarked() is called the variable queueLength of the
        //respective Mark object is updated accordingly. Therefore it must
        //be guaranteed that this method is only invoked once every time
        //method read() is called.
        boolean marked=m.isMarked(marks);
        if ((marked==false) && (m.isQueueEmpty())) {
            actualChar=isr.read();
        }
        else if (marked==true) {
            actualChar=isr.read();
            m.insertIntoQueue(actualChar);
        }
        else {
            actualChar=m.valueOfQueue();
            m.deleteFromQueue();
        }
    }
    catch (EmptyStackException e) {
        actualChar=isr.read();
    }
}

//The method mark() allows remembering a read position in the InputSteamReader.
//Subsequent reads are buffered in order to reread them again after method
//reset() is called. In order to allow that calls of mark() and their
//respective calls of reset() can be arbitrarily nested a stack is used
//to store mark objects.
private void mark() throws IOException {
    Mark m=new Mark();
    marks.push(m);
}

private void reset() throws IOException {
    Object o=marks.peek();
    Mark m=(Mark) o;
    m.turnOff();
}

private boolean isDelimiter() throws IOException {
    Object o;
    //Check whether an existing delimiter starts with the actual character.

```

```
if ((o=delimiters.get(new Character((char) actualChar))) != null) {
    mark();
    String candidate=new String(new char[] {(char) actualChar});
    int maxLength=1;
    Vector v=(Vector) o;
    Iterator it=v.iterator();
    //Iterate over all stored delimiters starting with the actual character
    //until one is found that matches the next d.getLength() characters in
    //the stream, which are stored in the String variable candidate. In order
    //to prevent rereading the same position in the stream several times an
    //int variable maxLength is used that indicates the maximal number of
    //read characters from the underlying stream.
    while (it.hasNext()) {
        Delimiter d=(Delimiter) it.next();
        int length=d.getLength();
        if (length<maxLength) {
            candidate=candidate.substring(0, length);
        }
        else {
            if (length>maxLength) {
                maxLength=length;
                for (int i=1; i<length; i++) {
                    //Ignore new line characters in a delimiter.
                    do {
                        read();
                    }
                    while ((actualChar==10) || (actualChar==13));
                    //Check whether the end of the file is reached.
                    if (actualChar != -1) {
                        candidate=candidate.concat(new String(new char[]
                                                                    {(char) actualChar}));
                    }
                    //If the end of the file is reached before d.getLength()
                    //characters could be read from the underlying stream the
                    //file obviously ends with a token.
                    else {
                        tokenDetected=true;
                        token=candidate;
                        endOfFile=true;
                        return false;
                    }
                }
            }
            else {
                candidate=candidate;
            }
        }
    }
    if (candidate.equals(d.getDelimiter())) {
        delimiter=candidate;
        //Read one (or in case of new line characters several) characters
        //ahead to determine the new actual character.
        do {
            read();
        }
        while ((actualChar==10) || (actualChar==13));
        if (actualChar == -1) {
            endOfFile=true;
        }
        return true;
    }
}
reset();
return false;
}
else {
```

```
        return false;
    }
}

private void getToken() throws IOException {
    do {
        token=token.concat(new String(new char[] {(char) actualChar}));
        //Ignore new line characters in a token.
        do {
            read();
        }
        while ((actualChar==10) || (actualChar==13));
    }
    while ((actualChar != -1) && (!isDelimiter()));
    if (!endOfFile) {
        delimiterDetected=true;
    }
    else {
        endOfFile=true;
    }
}
}
```

```

/*****
/* Author:      Stephanie Balzer
/* Project:     Development of a metadata driven tool to build the history of
/*              data in data warehouses
/* Copyright:   Department of Information Technology
/*              Database Technology Research Group
/*              University of Zurich, Switzerland
/* Language:    Java, Sun's Java Development Kit, Version 1.3.1
/* Class:       Delimiter
/* Version:     1.1
/* Date:        18.10.2001
/*-----
/* Description:
/*
/* A Delimiter allows an InputFileTokenizer to recognize the individual tokens.
/* Each instance of Delimiter stores the respective delimiter characters and
/* its length.
*****/

public class Delimiter {
    private int length;
    private String delimiter;

    public Delimiter(String delimiterString) {
        this.length=delimiterString.length();
        this.delimiter=delimiterString;
    }

    public String getDelimiter() {
        return this.delimiter;
    }

    public int getLength() {
        return this.length;
    }
}

```

```

/*****
/* Author:      Stephanie Balzer
/* Project:     Development of a metadata driven tool to build the history of
/*              data in data warehouses
/* Copyright:   Department of Information Technology
/*              Database Technology Research Group
/*              University of Zurich, Switzerland
/* Language:    Java, Sun's Java Development Kit, Version 1.3.1
/* Class:       Mark
/* Version:     1.0
/* Date:        08.10.2001
/*-----
/* Description:
/*
/* The class Mark is used by the InputFileTokenizer to remember a read position
/* in the InputStreamReader and to buffer subsequent reads. An instance of Mark
/* is created every time method mark() in class InputFileTokenizer is called.
/* Every Mark object maintains a queue to buffer subsequent reads.
*****/

import java.util.*;

public class Mark {
    //Contains all read characters after mark() is called.
    private Queue markedChars;

    //Set true every time method reset() in class InputFileTokenizer is called.
    private boolean marked;

    //Indicates the number of elements in the queue. Is incremented with
    //every element that is inserted into the Queue when method mark() in class
    //InputFileTokenizer is called and decremented when method reset() is invoked.
    private int queueLength;

    public Mark() {
        markedChars=new Queue();
        marked=true;
        queueLength=0;
    }

    public boolean isMarked(Stack marks) {
        if (marked) {
            queueLength++;
            return true;
        }
        else {
            queueLength--;
            if (queueLength>=0) {
                return false;
            }
            else {
                marks.pop();
                return isMarked(marks);
            }
        }
    }

    public void turnOff() {
        marked=false;
    }

    public boolean isQueueEmpty() {
        return markedChars.isEmpty();
    }
}

```

```
public void insertIntoQueue(int actualChar) {
    markedChars.insert(new Integer(actualChar));
}

public int valueOfQueue() {
    Integer i=(Integer) markedChars.value();
    return i.intValue();
}

public void deleteFromQueue() {
    markedChars.delete();
}
}
```

```

/*****
/* Author:      Stephanie Balzer
/* Project:     Development of a metadata driven tool to build the history of
/*              data in data warehouses
/* Copyright:   Department of Information Technology
/*              Database Technology Research Group
/*              University of Zurich, Switzerland
/* Language:    Java, Sun's Java Development Kit, Version 1.3.1
/* Class:       Queue
/* Version:     1.0
/* Date:        08.10.2001
/*-----
/* Description:
/*
/* Class Queue represents a first-in-first-out (FIFO) queue of objects.
*****/

import java.util.*;

public class Queue extends Vector{
    //Specifies the index of the Queue where the next Object has to be inserted.
    private static int nextIndex;

    public Queue() {
        nextIndex=0;
    }

    public void insert(Object element) {
        add(nextIndex, element);
        nextIndex++;
    }

    public void delete() {
        remove(0);
        nextIndex--;
    }

    public Object value() {
        return get(0);
    }
}

```



```

/*****
/* Author:      Stephanie Balzer
/* Project:     Development of a metadata driven tool to build the history of
/*              data in data warehouses
/* Copyright:   Department of Information Technology
/*              Database Technology Research Group
/*              University of Zurich, Switzerland
/* Language:    Java, Sun's Java Development Kit, Version 1.3.1
/* Class:       HistoryComponent
/* Version:     1.0
/* Date:        17.09.2001
/* -----
/* Description:
/*
/* HistoryComponent is an abstract class that is the superclass of all kinds of
/* HistComp's. For each dimension of the abstract history model HistoryComponent
/* defines a class variable. Every subclass has therefore to indicate its
/* respective type of building the history by setting the class variables
/* accordingly. Because of the restriction that all subclasses of
/* HistoryComponent are singletons every subclass must provide a class method
/* getInstance() which makes sure that only once a subclass is initialised and
/* returns that single instance. HistoryComponent also declares the abstract
/* methods insert(), update() and delete() which every subclass has to implement
/* according to its history type. The abstract method setEntity() is used to fix
/* the database entity to which the database operations are applied.
*****/

import java.io.*;
import java.util.*;
import java.sql.*;

abstract public class HistoryComponent {
    protected static String timeDeclaration;
    protected static String timeDimension;
    protected static String timeGranularity;
    protected static String dataGranularity;
    protected static String developmentSpecification;
    protected static String developmentProcess;
    protected static String directionOfGrowth;

    abstract public void setEntity(String name);

    abstract public void insert(Vector parameters) throws SQLException,
        TemporalFormatException, NumberOfArgumentException;
    abstract public void update(Vector parameters) throws SQLException,
        TemporalFormatException, NumberOfArgumentException;
    abstract public void delete(Vector parameters) throws SQLException,
        TemporalFormatException, NumberOfArgumentException;

    public void showHistoryType() {
        System.out.println("time declaration: " + timeDeclaration);
        System.out.println("time dimension: " + timeDimension);
        System.out.println("time granularity: " + timeGranularity);
        System.out.println("data granularity: " + dataGranularity);
        System.out.println("development specification: " + developmentSpecification);
        System.out.println("development process: " + developmentProcess);
        System.out.println("direction of growth: " + directionOfGrowth);
    }
}

```

```

/*****
/* Author:      Stephanie Balzer
/* Project:     Development of a metadata driven tool to build the history of
/*              data in data warehouses
/* Copyright:   Department of Information Technology
/*              Database Technology Research Group
/*              University of Zurich, Switzerland
/* Language:    Java, Sun's Java Development Kit, Version 1.3.1
/* Class:       HistCompRDB
/* Version:     1.0
/* Date:        17.09.2001
/*-----
/* Description:
/*
/* HistCompRDB is the direct subclass of HistoryComponent. It is also an
/* abstract class which is the superclass of all HistComp's that govern the
/* management of historical information of relational databases. HistCompRDB
/* defines the constants now and uc which are used to indicate values that are
/* not known yet respectively haven't changed yet. The class defines various
/* methods to convert String representations of time indications to the
/* appropriate format and methods to get respectively update Columns with
/* String literals.
*****/

import java.sql.*;
import java.text.*;
import java.io.*;
import java.math.*;

abstract public class HistCompRDB extends HistoryComponent{
    protected static String relName;
    //now and uc have the maximum possible value for ORACLE data type DATE.
    //now conforms to the Java class Date. uc conforms to the Java class Timestamp
    //with the exception that the nanoseconds are not indicated and thus
    //implicitly set to zero.
    protected static final String now="47121231";           //yyyymmdd
    protected static final String uc= "00010101000000";    //yyyymmddhhmmss

    public void setEntity(String name) {
        relName=name;
    }

    public void showEntityName() {
        if (relName!=null)
            System.out.println("relation name: " + relName);
        else
            System.out.println("relation name not yet indicated");
    }

    //Returns a String that can serve as input to java.sql.Date.valueOf().
    //The parameter time consists of numbers concatenated without punctuation
    //having the format yyyymmdd.
    public static String convertToDateFormat(String time)
        throws TemporalFormatException {
        if ((time.equals("")) || (time==null)) {
            throw new TemporalFormatException("method convertToDateFormat() needs " +
                "a date indication as input parameter");
        }
        else {
            if (time.equalsIgnoreCase("now")) {
                return convertToDateFormat(now);
            }
            else {
                StringCharacterIterator iterator=new StringCharacterIterator(time);

```

```

String date=new String();
int endIndex=iterator.getEndIndex();
if (endIndex-1 != 7) {
    throw new TemporalFormatException("invalid parameter '" + time +
                                      "' for method convertToDateFormat()");
}
else {
    for (int i=0; i<=3; i++) {
        date=date.concat(new Character(iterator.setIndex(i)).toString());
    }
    date=date.concat("-");
    for (int i=4; i<=5; i++) {
        date=date.concat(new Character(iterator.setIndex(i)).toString());
    }
    date=date.concat("-");
    for (int i=6; i<=7; i++) {
        date=date.concat(new Character(iterator.setIndex(i)).toString());
    }
}
return date;
}
}
}

```

```

//Returns a String that can serve as input to java.sql.Time.valueOf().
//The parameter inTime consists of numbers concatenated without punctuation
//having the format hhmmss. Missing seconds are accepted and set to zero.
public static String convertToTimeFormat(String inTime)
    throws TemporalFormatException {
    if ((inTime.equals("")) || (inTime==null)) {
        throw new TemporalFormatException("method convertToTimeFormat() needs " +
                                           "a time indication as input parameter");
    }
    else {
        StringCharacterIterator iterator=new StringCharacterIterator(inTime);
        String time=new String();
        int endIndex=iterator.getEndIndex();
        if ((endIndex-1 < 3) || (endIndex-1 > 5)) {
            throw new TemporalFormatException("invalid parameter '" + inTime +
                                              "' for method convertToTimeFormat()");
        }
        else {
            for (int i=0; i<=1; i++) {
                time=time.concat(new Character(iterator.setIndex(i)).toString());
            }
            time=time.concat(":");
            for (int i=2; i<=3; i++) {
                time=time.concat(new Character(iterator.setIndex(i)).toString());
            }
            if (endIndex-1==3) {
                return time=time.concat(":00");
            }
            else {
                if (endIndex-1<5) {
                    throw new TemporalFormatException("invalid parameter '" +
                                                       inTime + "' for method convertToTimeFormat(): seconds " +
                                                       "are not correctly indicated");
                }
                else {
                    time=time.concat(":");
                    for (int i=4; i<=5; i++) {
                        time=time.concat(
                            new Character(iterator.setIndex(i)).toString());
                    }
                }
            }
        }
    }
}

```

```

        }
        return time;
    }
}

//Returns a String that can serve as input to java.sql.Timestamp.valueOf().
//The parameter time consists of numbers concatenated without punctuation
//having the format yyyyymmddhhmmssffffff. Missing time part or missing
//seconds or nanoseconds are accepted.
public static String convertToTimestampFormat(String time)
    throws TemporalFormatException {
    if ((time.equals("")) || (time==null)) {
        throw new TemporalFormatException("method convertToTimestampFormat()" +
            "needs a timestamp indication as input parameter");
    }
    else {
        if (time.equalsIgnoreCase("uc")) {
            return convertToTimestampFormat(uc);
        }
        else {
            StringCharacterIterator iterator=new StringCharacterIterator(time);
            String timestamp=new String();
            int endIndex=iterator.getEndIndex();
            if ((endIndex-1 < 7) || (endIndex-1 > 22)) {
                throw new TemporalFormatException("invalid parameter '" + time +
                    "' for method convertToTimestampFormat()");
            }
            else {
                for (int i=0; i<=3; i++) {
                    timestamp=timestamp.concat(
                        new Character(iterator.setIndex(i)).toString());
                }
                timestamp=timestamp.concat("-");
                for (int i=4; i<=5; i++) {
                    timestamp=timestamp.concat(
                        new Character(iterator.setIndex(i)).toString());
                }
                timestamp=timestamp.concat("-");
                for (int i=6; i<=7; i++) {
                    timestamp=timestamp.concat(
                        new Character(iterator.setIndex(i)).toString());
                }
                if (endIndex-1 == 7) {
                    return timestamp=timestamp.concat(" 00:00:00.000000000");
                }
                else {
                    if (endIndex-1 < 11) {
                        throw new TemporalFormatException("invalid parameter '" +
                            time + "' for method convertToTimestampFormat(): " +
                                "time part is not correctly indicated");
                    }
                    else {
                        timestamp=timestamp.concat(" ");
                        for (int i=8; i<=9; i++) {
                            timestamp=timestamp.concat(
                                new Character(iterator.setIndex(i)).toString());
                        }
                        timestamp=timestamp.concat(":");
                        for (int i=10; i<=11; i++) {
                            timestamp=timestamp.concat(
                                new Character(iterator.setIndex(i)).toString());
                        }
                        if (endIndex-1 == 11) {
                            return timestamp=timestamp.concat(":00.000000000");
                        }
                    }
                }
            }
        }
    }
}

```



```

//chunks are getBinaryStream() an updateBinaryStream() cannot be used.
case -4: return rs.getBytes(index);
//VARBINARY:
case -3: return rs.getBytes(index);
//BINARY:
case -2: return rs.getBytes(index);
//LONGVARCHAR: Because one does not know at this point how long the
//chunks are getAsciiStream() and updateAsciiStream() cannot be used.
case -1: return rs.getString(index);
//CHAR:
case 1: return rs.getString(index);
//NUMERIC:
case 2: return rs.getBigDecimal(index);
//DECIMAL:
case 3: return rs.getBigDecimal(index);
//INTEGER:
case 4: return new Integer(rs.getInt(index));
//SMALLINT:
case 5: return new Short(rs.getShort(index));
//FLOAT:
case 6: return new Double(rs.getDouble(index));
//REAL:
case 7: return new Float(rs.getFloat(index));
//DOUBLE:
case 8: return new Double(rs.getDouble(index));
//VARCHAR:
case 12: return rs.getString(index);
//DATE:
case 91: return rs.getDate(index);
//TIME:
case 92: return rs.getTime(index);
//TIMESTAMP:
case 93: return rs.getTimestamp(index);
default: throw new SQLException("Column value is possible not " +
    "correctly read from the database because no adequate " +
    "getXXX() method was found.");
    }
}

public static void updateColumnValue(ResultSet rs, int index, Object object)
    throws SQLException {
    ResultSetMetaData rsmd=rs.getMetaData();
    switch (rsmd.getColumnType(index)) {
        //BIT:
        case -7: rs.updateBoolean(index, ((Boolean) object).booleanValue());
            break;
        //TINYINT:
        case -6: rs.updateByte(index, ((Byte) object).byteValue());
            break;
        //BIGINT:
        case -5: rs.updateLong(index, ((Long) object).longValue());
            break;
        //LONGVARBINARY: Because one does not know at this point how long the
        //chunks are getBinaryStream() an updateBinaryStream() cannot be used.
        case -4: rs.updateBytes(index, (byte[]) object);
            break;
        //VARBINARY:
        case -3: rs.updateBytes(index, (byte[]) object);
            break;
        //BINARY:
        case -2: rs.updateBytes(index, (byte[]) object);
            break;
        //LONGVARCHAR: Because one does not know at this point how long the chunks
        //are getAsciiStream() and updateAsciiStream() cannot be used.
        case -1: rs.updateString(index, (String) object);
    }
}

```

```

        break;
//CHAR:
case 1: rs.updateString(index, (String) object);
        break;
//NUMERIC:
case 2: rs.updateBigDecimal(index, (BigDecimal) object);
        break;
//DECIMAL:
case 3: rs.updateBigDecimal(index, (BigDecimal) object);
        break;
//INTEGER:
case 4: rs.updateInt(index, ((Integer) object).intValue());
        break;
//SMALLINT:
case 5: rs.updateShort(index, ((Short) object).shortValue());
        break;
//FLOAT:
case 6: rs.updateDouble(index, ((Double) object).doubleValue());
        break;
//REAL:
case 7: rs.updateFloat(index, ((Float) object).floatValue());
        break;
//DOUBLE:
case 8: rs.updateDouble(index, ((Double) object).doubleValue());
        break;
//VARCHAR:
case 12: rs.updateString(index, (String) object);
        break;
//DATE:
case 91: rs.updateDate(index, (Date) object);
        break;
//TIME:
case 92: rs.updateTime(index, (Time) object);
        break;
//TIMESTAMP:
case 93: rs.updateTimestamp(index, (Timestamp) object);
        break;
default: throw new SQLException("Column value is possible not " +
        "correctly written to the database because no adequate " +
        "updateXXX() method was found.");
    }
}

public static void updateColumnValue(ResultSet rs, int index, String value)
        throws SQLException, TemporalFormatException {
    ResultSetMetaData rsmd=rs.getMetaData();
    switch (rsmd.getColumnType(index)) {
        //BIT:
        case -7: rs.updateBoolean(index, Boolean.valueOf(value).booleanValue());
                break;
        //TINYINT:
        case -6: rs.updateByte(index, Byte.parseByte(value));
                break;
        //BIGINT:
        case -5: rs.updateLong(index, Long.parseLong(value));
                break;
        //LONGVARBINARY: Because one does not know at this point how long the
        //chunks are getBinaryStream() an updateBinaryStream() cannot be used.
        case -4: rs.updateBytes(index, value.getBytes());
                break;
        //VARBINARY:
        case -3: rs.updateBytes(index, value.getBytes());
                break;
        //BINARY:
        case -2: rs.updateBytes(index, value.getBytes());
    }
}

```

```

        break;
//LONGVARCHAR: Because one does not know at this point how long the chunks
//are getAsciiStream() and updateAsciiStream() cannot be used.
case -1: rs.updateString(index, value);
        break;
//CHAR:
case 1: rs.updateString(index, value);
        break;
//NUMERIC:
case 2: rs.updateBigDecimal(index, new BigDecimal(value));
        break;
//DECIMAL:
case 3: rs.updateBigDecimal(index, new BigDecimal(value));
        break;
//INTEGER:
case 4: rs.updateInt(index, Integer.parseInt(value));
        break;
//SMALLINT:
case 5: rs.updateShort(index, Short.parseShort(value));
        break;
//FLOAT:
case 6: rs.updateDouble(index, Double.parseDouble(value));
        break;
//REAL:
case 7: rs.updateFloat(index, Float.parseFloat(value));
        break;
//DOUBLE:
case 8: rs.updateDouble(index, Double.parseDouble(value));
        break;
//VARCHAR:
case 12: rs.updateString(index, value);
        break;
//DATE:
case 91: rs.updateDate(index, Date.valueOf(convertToDateFormat(value)));
        break;
//TIME:
case 92: rs.updateTime(index, Time.valueOf(convertToTimeFormat(value)));
        break;
//TIMESTAMP:
case 93: rs.updateTimestamp(index, Timestamp.valueOf(
            convertToTimestampFormat(value)));
        break;
default: throw new SQLException("Column value is possible not " +
    "correctly written to the database because no adequate " +
    "updateXXX() method was found.");
    }
}
}

```



```

/*****
/* Author:      Stephanie Balzer
/* Project:     Development of a metadata driven tool to build the history of
/*              data in data warehouses
/* Copyright:   Department of Information Technology
/*              Database Technology Research Group
/*              University of Zurich, Switzerland
/* Language:    Java, Sun's Java Development Kit, Version 1.3.1
/* Class:       HistCompRDB_1
/* Version:     1.3
/* Date:        26.09.2001
/*-----
/* Description:
/*
/* HistCompRDB_1 is a direct subclass of HistCompRDB and accomplishes the
/* following kind of building the history: explicit time declaration, uses valid
/* time and transaction time, indicates valid time in date granularity and
/* transaction time in timestamp granularity, data granularity is the tuple,
/* development specification is complete, development process is state interval,
/* direction of growth is vertical. HistCompRDB_1 implements the methods
/* insert(), update() and delete() and provides method getInstance() to get the
/* singleton design pattern through.
*****/

import java.io.*;
import java.sql.*;
import java.math.*;
import java.util.Vector;

public class HistCompRDB_1 extends HistCompRDB {
    private static HistCompRDB_1 hc;

    private HistCompRDB_1() {
        timeDeclaration="explicit";
        timeDimension="valid time and transaction time";
        timeGranularity="date for valid time, timestamp for transaction time";
        dataGranularity="tuple";
        developmentSpecification="complete";
        developmentProcess="state interval";
        directionOfGrowth="vertical";
    }

    public static HistCompRDB_1 getInstance() {
        if (hc==null)
            return hc=new HistCompRDB_1();
        else
            return hc;
    }

    public void insert(Vector parameters) throws SQLException,
        TemporalFormatException,
        NumberOfArgumentException {

        int numberOfParameters=parameters.size();
        try {
            Statement stmt=(Statement) parameters.get(0);
            Vector vNames=(Vector) parameters.get(1);
            Vector vValues=(Vector) parameters.get(2);
            int nameLength=vNames.size();
            int valueLength=vValues.size();
            if (nameLength != valueLength) {
                throw new NumberOfArgumentException("To insert a new record the " +
                    "number of indicated attribute names should be equal to the " +
                    "number of provided values. The given attributes and values " +
                    "are:\n" + vNames.toString() + "\n" + vValues.toString());
            }
        }
    }
}

```

```

    }
    String[] names=(String[]) vNames.toArray(new String[nameLength]);
    String[] values=(String[]) vValues.toArray(new String[valueLength]);
    String VST=(String) parameters.get(3);
    String TST=(String) parameters.get(4);
    if (numberOfParameters==5) {
        insert(stmt, names, values, VST, TST);
    }
    else {
        String VET=(String) parameters.get(5);
        insert(stmt, names, values, VST, TST, VET);
    }
}
catch (ClassCastException e) {
    throw new NumberOfArgumentException("Method insert() was invoked with " +
        "the following, incorrect number of arguments:\n" +
        parameters.toString());
}
catch (ArrayIndexOutOfBoundsException e) {
    throw new NumberOfArgumentException("Method insert() was invoked with " +
        "the following, incorrect number of arguments:\n" +
        parameters.toString());
}
}

public void update (Vector parameters) throws SQLException,
    TemporalFormatException,
    NumberOfArgumentException {
    int numberOfParameters=parameters.size();
    try {
        Statement stmt=(Statement) parameters.get(0);
        String ntPKeyName=(String) parameters.get(1);
        String ntPKeyValue=(String) parameters.get(2);
        Vector vNames=(Vector) parameters.get(3);
        Vector vValues=(Vector) parameters.get(4);
        int nameLength=vNames.size();
        int valueLength=vValues.size();
        if (nameLength != valueLength) {
            throw new NumberOfArgumentException("To update a record the number " +
                "of indicated attribute names should be equal to the number of " +
                "provided values. The given attributes and values are:\n" +
                vNames.toString() + "\n" + vValues.toString());
        }
        String[] names=(String[]) vNames.toArray(new String[nameLength]);
        String[] values=(String[]) vValues.toArray(new String[valueLength]);
        String oldVET=(String) parameters.get(5);
        String newVST=(String) parameters.get(6);
        String newTT=(String) parameters.get(7);
        if (numberOfParameters==8) {
            update(stmt, ntPKeyName, ntPKeyValue, names, values, oldVET,
                newVST, newTT);
        }
        else {
            String newVET=(String) parameters.get(8);
            update(stmt, ntPKeyName, ntPKeyValue, names, values, oldVET,
                newVST, newTT, newVET);
        }
    }
    catch (ClassCastException e) {
        throw new NumberOfArgumentException("Method update() was invoked with " +
            "the following, incorrect number of arguments:\n" +
            parameters.toString());
    }
    catch (ArrayIndexOutOfBoundsException e) {
        throw new NumberOfArgumentException("Method update() was invoked with " +

```

```

        "the following, incorrect number of arguments:\n" +
        parameters.toString());
    }
}

public void delete (Vector parameters) throws SQLException,
    TemporalFormatException,
    NumberOfArgumentException {
    int numberOfParameters=parameters.size();
    try {
        Statement stmt=(Statement) parameters.get(0);
        String ntPKeyName=(String) parameters.get(1);
        String ntPKeyValue=(String) parameters.get(2);
        String VET=(String) parameters.get(3);
        String TT=(String) parameters.get(4);
        delete(stmt, ntPKeyName, ntPKeyValue, VET, TT);
    }
    catch (ClassCastException e) {
        throw new NumberOfArgumentException("Method delete() was invoked with " +
            "the following, incorrect number of arguments:\n" +
            parameters.toString());
    }
    catch (ArrayIndexOutOfBoundsException e) {
        throw new NumberOfArgumentException("Method delete() was invoked with " +
            "the following, incorrect number of arguments:\n" +
            parameters.toString());
    }
}

//This method is invoked if valid end time (VET) is not known at insert time
//and therefore now.
private void insert(Statement stmt, String[] names, String[] values,
    String VST, String TST)
    throws SQLException, TemporalFormatException {
    ResultSet rs=stmt.executeQuery("select " + getAllColumns(stmt) +
        " from " + relName);

    rs.moveToInsertRow();
    int numberOfChgdColumns=names.length;
    int index;
    for (int i=0; i<numberOfChgdColumns; i++) {
        index=rs.findColumn(names[i]);
        updateColumnValue(rs, index, values[i]);
    }
    rs.updateDate("VST", Date.valueOf(convertToDateFormat(VST)));
    rs.updateDate("VET", Date.valueOf(convertToDateFormat(now)));
    rs.updateTimestamp("TST", Timestamp.valueOf(convertToTimestampFormat(TST)));
    rs.updateTimestamp("TET", Timestamp.valueOf(convertToTimestampFormat(uc)));
    rs.insertRow();
    rs.close();
}

//This method is invoked if valid end time (VET) is known at insert time.
private void insert(Statement stmt, String[] names, String[] values, String VST,
    String TST, String VET) throws SQLException,
    TemporalFormatException {
    ResultSet rs=stmt.executeQuery("select " + getAllColumns(stmt) +
        " from " + relName);

    rs.moveToInsertRow();
    int numberOfChgdColumns=names.length;
    int index;
    for (int i=0; i<numberOfChgdColumns; i++) {
        index=rs.findColumn(names[i]);
        updateColumnValue(rs, index, values[i]);
    }
    rs.updateDate("VST", Date.valueOf(convertToDateFormat(VST)));

```

```

rs.updateDate("VET", Date.valueOf(convertToDateFormat(VET)));
rs.updateTimestamp("TST", Timestamp.valueOf(convertToTimestampFormat(TST)));
rs.updateTimestamp("TET", Timestamp.valueOf(convertToTimestampFormat(uc)));
rs.insertRow();
rs.close();
}

//This method is invoked if valid end time (newVET) is not known
//at insert time and therefore now.
private void update(Statement stmt, String ntPKeyName, String ntPKeyValue,
String[] names, String[] values, String oldVET,
String newVST, String newTT)
throws SQLException, TemporalFormatException {
//Find valid end time value of the tuple that has to be updated:
ResultSet rs=stmt.executeQuery("select max(VET) from " + relName +
" where " + ntPKeyName + "=" + ntPKeyValue +
" and TET=to_date('" + uc + "', 'YYYYMMDDHH24MISS')");

rs.next();
String maxVET;
try {
maxVET=rs.getDate(1).toString();
}
catch (NullPointerException e) {
throw new SQLException("The tuple with the nontemporal primary key " +
"name '" + ntPKeyName + "' and the nontemporal primary key value '" +
ntPKeyValue + "' does not exist and can therefore not be updated!");
}
//Retrieve the tuple that has to be updated:
rs=stmt.executeQuery("select " + getAllColumns(stmt) + " from " + relName +
" where " + ntPKeyName + "=" + ntPKeyValue +
" and TET=to_date('" + uc + "', 'YYYYMMDDHH24MISS') " +
"and VET=to_date('" + maxVET + "', 'YYYY-MM-DD')");
//Make a copy of the tuple that has to be updated:
rs.next();
ResultSetMetaData rsmd=rs.getMetaData();
int numberOfColumns=rsmd.getColumnCount();
Object[] objects=new Object[numberOfColumns];
Object object;
for (int i=0; i<numberOfColumns; i++) {
object=getColumnValue(rs, i+1);
objects[i]=object;
}
rs.moveToInsertRow();
for (int i=0; i<numberOfColumns; i++) {
updateColumnValue(rs, i+1, objects[i]);
}
//Set valid end time and transaction start time of that copy and insert
//the copy as a new tuple into the relation:
rs.updateDate("VET", Date.valueOf(convertToDateFormat(oldVET)));
rs.updateTimestamp("TST",
Timestamp.valueOf(convertToTimestampFormat(newTT)));
rs.insertRow();
//Set the transaction end time of the original tuple, that has been copied:
rs.moveToCurrentRow();
rs.updateTimestamp("TET",
Timestamp.valueOf(convertToTimestampFormat(newTT)));
rs.updateRow();
//Retrieve the newly inserted tuple and make a copy of it:
rs=stmt.executeQuery("select " + getAllColumns(stmt) + " from " + relName +
" where " + ntPKeyName + "=" + ntPKeyValue +
" and TET=to_date('" + uc + "', 'YYYYMMDDHH24MISS') " +
"and VET=to_date('" + oldVET + "', 'YYYYMMDD')");

rs.next();
for (int i=0; i<numberOfColumns; i++) {
object=getColumnValue(rs, i+1);

```

```

        objects[i]=object;
    }
    rs.moveToInsertRow();
    for (int i=0; i<numberOfColumns; i++) {
        updateColumnValue(rs, i+1, objects[i]);
    }
    //Update the respective columns of that copy and set valid end time and
    //transaction start time. Finally insert the copy as a new tuple into
    //the relation:
    int numberOfChgdColumns=names.length;
    int index;
    for (int i=0; i<numberOfChgdColumns; i++) {
        index=rs.findColumn(names[i]);
        updateColumnValue(rs, index, values[i]);
    }
    rs.updateDate("VST", Date.valueOf(convertToDateFormat(newVST)));
    rs.updateDate("VET", Date.valueOf(convertToDateFormat(now)));
    rs.insertRow();
    rs.close();
}

//This method is invoked if valid end time (newVET) is known at update time.
private void update(Statement stmt, String ntPKeyName, String ntPKeyValue,
                    String[] names, String[] values, String oldVET,
                    String newVST, String newTT, String newVET)
    throws SQLException, TemporalFormatException {
    //Find valid end time value of the tuple that has to be updated:
    ResultSet rs=stmt.executeQuery("select max(VET) from " + relName +
                                    " where " + ntPKeyName + "=" + ntPKeyValue +
                                    "' and TET=to_date('" + uc +
                                    "','YYYYMMDDHH24MISS')");

    rs.next();
    String maxVET;
    try {
        maxVET=rs.getDate(1).toString();
    }
    catch (NullPointerException e) {
        throw new SQLException("The tuple with the nontemporal primary key " +
                                "name '" + ntPKeyName + "' and the nontemporal primary key value '" +
                                ntPKeyValue + "' does not exist and can therefore not be updated!");
    }
    //Retrieve the tuple that has to be updated:
    rs=stmt.executeQuery("select " + getAllColumns(stmt) + " from " + relName +
                          " where " + ntPKeyName + "=" + ntPKeyValue +
                          "' and TET=to_date('" + uc + "','YYYYMMDDHH24MISS') and " +
                          "VET=to_date('" + maxVET + "','YYYY-MM-DD')");
    //Make a copy of the tuple that has to be updated:
    rs.next();
    ResultSetMetaData rsmd=rs.getMetaData();
    int numberOfColumns=rsmd.getColumnCount();
    Object[] objects=new Object[numberOfColumns];
    Object object;
    for (int i=0; i<numberOfColumns; i++) {
        object=getColumnValue(rs, i+1);
        objects[i]=object;
    }
    rs.moveToInsertRow();
    for (int i=0; i<numberOfColumns; i++) {
        updateColumnValue(rs, i+1, objects[i]);
    }
    //Set valid end time and transaction start time of that copy and insert
    //the copy as a new tuple into the relation:
    rs.updateDate("VET", Date.valueOf(convertToDateFormat(oldVET)));
    rs.updateTimestamp("TST", Timestamp.valueOf(
        convertToTimestampFormat(newTT)));

```

```

rs.insertRow();
//Set the transaction end time of the original tuple, that has been copied:
rs.moveToCurrentRow();
rs.updateTimestamp("TET", Timestamp.valueOf(
    convertToTimestampFormat(newTT)));
rs.updateRow();
//Retrieve the newly inserted tuple and make a copy of it:
rs=stmt.executeQuery("select " + getAllColumns(stmt) + " from " + relName +
    " where " + ntPKeyName + "=" + ntPKeyValue +
    "' and TET=to_date('" + uc + "','YYYYMMDDHH24MISS') and " +
    "VET=to_date('" + oldVET + "','YYYYMMDD')");
rs.next();
for (int i=0; i<numberOfColumns; i++) {
    object=getColumnValue(rs, i+1);
    objects[i]=object;
}
rs.moveToInsertRow();
for (int i=0; i<numberOfColumns; i++) {
    updateColumnValue(rs, i+1, objects[i]);
}
//Update the respective columns of that copy and set valid end time and
//transaction start time. Finally insert the copy as a new tuple into
//the relation:
int numberOfChgdColumns=names.length;
int index;
for (int i=0; i<numberOfChgdColumns; i++) {
    index=rs.findColumn(names[i]);
    updateColumnValue(rs, index, values[i]);
}
rs.updateDate("VST", Date.valueOf(convertToDateFormat(newVST)));
rs.updateDate("VET", Date.valueOf(convertToDateFormat(newVET)));
rs.insertRow();
rs.close();
}

private void delete(Statement stmt, String ntPKeyName, String ntPKeyValue,
    String VET, String TT) throws SQLException,
    TemporalFormatException {
    //Find valid end time value of the tuple that has to be deleted:
    ResultSet rs=stmt.executeQuery("select max(VET) from " + relName +
        " where " + ntPKeyName + "=" + ntPKeyValue +
        "' and TET=to_date('" + uc + "','YYYYMMDDHH24MISS')");
    rs.next();
    String maxVET;
    try {
        maxVET=rs.getDate(1).toString();
    }
    catch (NullPointerException e) {
        throw new SQLException("The tuple with the nontemporal primary key " +
            "name '" + ntPKeyName + "' and the nontemporal primary key value '" +
            ntPKeyValue + "' does not exist and can therefore not be deleted!");
    }
    //Retrieve the tuple that has to be deleted:
    rs=stmt.executeQuery("select " + getAllColumns(stmt) + " from " + relName +
        " where " + ntPKeyName + "=" + ntPKeyValue +
        "' and TET=to_date('" + uc + "','YYYYMMDDHH24MISS') and " +
        "VET=to_date('" + maxVET + "','YYYY-MM-DD')");
    //Make a copy of the tuple that has to be deleted:
    rs.next();
    ResultSetMetaData rsmd=rs.getMetaData();
    int numberOfColumns=rsmd.getColumnCount();
    Object[] objects=new Object[numberOfColumns];
    Object object;
    for (int i=0; i<numberOfColumns; i++) {
        object=getColumnValue(rs, i+1);

```

```
        objects[i]=object;
    }
    rs.moveToInsertRow();
    for (int i=0; i<numberOfColumns; i++) {
        updateColumnValue(rs, i+1, objects[i]);
    }
    //Set valid end time, transaction start and end time of that copy and
    //insert the copy as a new tuple into the relation:
    rs.updateDate("VET", Date.valueOf(convertToDateFormat(VET)));
    rs.updateTimestamp("TST", Timestamp.valueOf(convertToTimestampFormat(TT)));
    rs.updateTimestamp("TET", Timestamp.valueOf(convertToTimestampFormat(uc)));
    rs.insertRow();
    //Set the transaction end time of the original tuple, that has been copied:
    rs.moveToCurrentRow();
    rs.updateTimestamp("TET", Timestamp.valueOf(convertToTimestampFormat(TT)));
    rs.updateRow();
    rs.close();
}
}
```

```

/*****
/* Author:      Stephanie Balzer
/* Project:     Development of a metadata driven tool to build the history of
/*              data in data warehouses
/* Copyright:   Department of Information Technology
/*              Database Technology Research Group
/*              University of Zurich, Switzerland
/* Language:    Java, Sun's Java Development Kit, Version 1.3.1
/* Class:       NumberOfArgumentException
/* Version:     1.0
/* Date:        19.10.2001
/*-----
/* Description:
/*
/* A NumberOfArgumentException is thrown to indicate that something with the
/* number or type of the parameters passed to a method is wrong.
*****/

public class NumberOfArgumentException extends Exception {

    public NumberOfArgumentException() {
        super();
    }

    public NumberOfArgumentException(String message) {
        super(message);
    }
}

```



```

/*****
/* Author:      Stephanie Balzer
/* Project:     Development of a metadata driven tool to build the history of
/*              data in data warehouses
/* Copyright:   Department of Information Technology
/*              Database Technology Research Group
/*              University of Zurich, Switzerland
/* Language:    Java, Sun's Java Development Kit, Version 1.3.1
/* Class:       DeltaFileException
/* Version:     1.0
/* Date:        18.10.2001
/*-----
/* Description:
/*
/* A DeltaFileException is thrown to indicate an unusual or faulty structure of
/* the delta file.
*****/

public class DeltaFileException extends Exception {

    public DeltaFileException() {
        super();
    }

    public DeltaFileException(String message) {
        super(message);
    }
}

```

```

/*****
/* Author:      Stephanie Balzer
/* Project:     Development of a metadata driven tool to build the history of
/*              data in data warehouses
/* Copyright:   Department of Information Technology
/*              Database Technology Research Group
/*              University of Zurich, Switzerland
/* Language:    Java, Sun's Java Development Kit, Version 1.3.1
/* Class:       TemporalFormatException
/* Version:     1.0
/* Date:        18.10.2001
/*-----
/* Description:
/*
/* A TemporalFormatException is thrown to indicate that the time values
/* supplied by the delta file are not in the appropriate format.
*****/

public class TemporalFormatException extends Exception {

    public TemporalFormatException() {
        super();
    }

    public TemporalFormatException(String message) {
        super(message);
    }
}

```

LITERATURVERZEICHNIS

- [BaGü] A. Bauer, H. Günzel. Data-Warehouse-Systeme: Architektur, Entwicklung, Anwendung. dpunkt.verlag, 2001.
- [BJSS] R. Bliujute, C.S. Jensen, S. Saltenis, G. Slivinskas. Systematic Change Management in Dimensional Data Warehousing, TR-23. A TimeCenter Technical Report, 1998.
<http://www.cs.auc.dk/TimeCenter>.
- [Catt] R. Cattell. Object Data Management: Object-Oriented and Extended Relational Database Systems. Addison-Wesley, 1994.
- [ChDa] S. Chaudhuri, U. Dayal. An Overview of Data Warehousing and OLAP-Technology. SIGMOD Record 26(1), March 1997.
- [Codd] E.F. Codd. The Relational Model for Database Management, Version 2. Addison-Wesley, 1990.
- [DevI] B. Devlin. Data Warehouse: From Architecture to Implementation. Addison-Wesley, 1997.
- [Ditt] K.R. Dittrich et al. Data Warehousing - was Sie schon immer darüber wissen wollten. Institut für Informatik der Universität Zürich, 2000.
- [DiVa] K.R. Dittrich, A. Vaduva. Metadata Management for Data Warehousing: Between Vision and Reality. Institut für Informatik der Universität Zürich, 2001.
<http://www.ifi.unizh.ch/dbtg/DBLibrary/index.html>.
- [DKVZ] K.R. Dittrich, J.-U. Kietz, A. Vaduva, R. Zücker. M⁴-The Mining Mart Meta Model. Institut für Informatik der Universität Zürich, 2001.
<http://www.ifi.unizh.ch/dbtg/DBLibrary/index.html>.
- [ElNa] R. Elmasri, S.B. Navathe. Fundamentals of Database Systems, third Edition. Addison-Wesley, 2000.
- [FoSc] M. Fowler, K. Scott. UML Distilled: Applying the Standard Object Modeling Language. Addison-Wesley, 1997.
- [GHJV] E. Gamma, R. Helm, R. Johnson, J. Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.
- [GÖS] I.A. Goralwalla, M.T. Özsu, D. Szafron. An Object-Oriented Framework for Temporal Data Models. Aus: O. Etzion, S. Jajodia, S. Sripada (Eds.). Temporal Databases: Research and Practice. Springer, 1998.
- [Inmo1] W.H. Inmon. Building the Data Warehouse, second Edition. John Wiley & Sons, 1996.
- [Inmo2] W.H. Inmon. Meta Data in the Data Warehouse: a Statement of Vision. 1997.
<http://www.billinmon.com/library/whiteprs/techtopic/tt10.pdf>.
- [Inmo3] W.H. Inmon. Meta Data in the Data Warehouse. 2000.
<http://www.billinmon.com/library/whiteprs/earlywp/ttmeta.pdf>.
- [Inmo4] W.H. Inmon. Time-Variant Data Structures. 2000.
<http://www.billinmon.com/library/whiteprs/earlywp/tds.pdf>.
- [Kimb1] R. Kimball. The Data Warehouse Toolkit: Practical Techniques for Building Dimensional Data Warehouses. John Wiley & Sons, 1996.
- [Kimb2] R. Kimball et al. The Data Warehouse Lifecycle Toolkit: Expert Methods for Designing, Developing, and Deploying Data Warehouses. John Wiley & Sons, 1999.

- Designing, Developing, and Deploying Data Warehouses. John Wiley & Sons, 1998.
- [Kimb3] R. Kimball. Slowly Changing Dimensions. DBMS online, 1996.
<http://www.dbmsmag.com/9604d05.html>.
- [Kimb4] R. Kimball. When a Slowly Changing Dimension Speeds up. Intelligent Enterprise Magazine, 1999.
http://www.iemagazine.com/db_area/archives/1999/990308/warehouse.shtml.
- [Kimb5] R. Kimball. It's Time for Time. DBMS online, 1997.
<http://www.dbmsmag.com/9707d05.html>.
- [KoLo] G. Koch, K. Loney. Oracle8i: The Complete Referene. McGraw-Hill, 2000.
- [Lued] J. Luedtke. Implementing Slowly Changing Dimensions. SQL Server Magazine, 2000.
<http://www.sqlmag.com/Articles/Index.cfm?ArticleID=7835>.
- [Netz] A. Netz. OLAP Services: Managing Slowly Changing Dimensions. Microsoft Corporation, 1999. <http://msdn.microsoft.com/library/techart/slowly2.htm>.
- [QGF] T. Quinlan, R. Gilbert, M. Ferguson. Modeling History for the Data Warehouse. Database P&D Online. <http://www.dbpd.com/vault/9811/quin.shtml>.
- [Schä] L. Schäfers. Temporal Modeling in Relational Database Systems. Verlag Dr. Kovač, 1991.
- [Schr] T. Schraml. Retyping Slowly Changing Dimensions. DM Review, 2000.
<http://www.dmreview.com/master.cfm?NavID=198&EdID=2263>.
- [Ste] A. Steiner. A Generalisation Approach to Temporal Data Models and their Implementations. Diss. ETH NO. 12434, 1998. <http://www.timeconsult.com>.

ABBILDUNGSVERZEICHNIS

ABBILDUNG 2-1: ARCHITEKTUR EINES DATA-WAREHOUSE-SYSTEMS, QUELLE: [DITT]	6
ABBILDUNG 2-2: MULTIDIMENSIONALES DATENMODELL	10
ABBILDUNG 2-3: HIERARCHISCHE ORDNUNG DER DIMENSIONSELEMENTE UND KONSOLIDIERUNGSPFADE	11
ABBILDUNG 2-4: STAR-SCHEMA	12
ABBILDUNG 2-5: METADATENVERWALTUNG IM DATA WAREHOUSING, QUELLE: [DITT]	13
ABBILDUNG 3-1: ORTHOGONALITÄT VON GÜLTIGKEITS- UND TRANSAKTIONSZEIT	17
ABBILDUNG 3-2: NICHT-TEMPORALES ER-DIAGRAMM FÜR DIE BEISPIELANWENDUNG	22
ABBILDUNG 3-3: SCHATTENZEITRAUM BEI EINER RETROAKTIVEN MODIFIKATION	31
ABBILDUNG 3-4: MODIFIKATIONSOPERATION INNERHALB EINES SCHATTENZEITRAUMES	31
ABBILDUNG 3-5: REFERENTIELLE INTEGRITÄT BEI TEMPORALER ERWEITERUNG	33
ABBILDUNG 3-6: EXTENSION ZUM STAR-SCHEMA IN ABBILDUNG 2-4	36
ABBILDUNG 3-7: ABSTRAKTES HISTORISIERUNGSMODELL	42
ABBILDUNG 4-1: METADATENSHEMA	46
ABBILDUNG 4-2: ARCHITEKTUR EINES METADATENGESTEUERTEN HISTORISIERUNGS-SYSTEMS	48
ABBILDUNG 5-1: UML-KLASSENDIAGRAMM DER JAVA-APPLIKATION	50
ABBILDUNG 5-2: DELTA-DATEI DES PROTOTYPS	54
ABBILDUNG 5-3: METHODEN ZUR DYNAMISCHEN, METADATENGESTEUERTEN KLASSEN- INSTANZIERUNG	56

TABELLENVERZEICHNIS

TABELLE 2-1: OPERATIVE VERSUS ANALYTISCHE DATENBANKEN, QUELLE: [DITT]	4
TABELLE 3-1: NICHT-TEMPORALES RELATIONENSHEMA FÜR DIE BEISPIELANWENDUNG	22
TABELLE 3-2: GÜLTIGKEITSZEIT MIT ZEITINTERVALLEN.....	23
TABELLE 3-3: NACHTEILE DER TUPEL-VERSIONIERUNG BEI ÄNDERUNG DER ATTRIBUTWERTE.....	24
TABELLE 3-4: EXPLIZITE HISTORIE DER ATTRIBUTWERTE	25
TABELLE 3-5: GÜLTIGKEITSZEIT MIT MOMENTAUFNAHMEN	26
TABELLE 3-6: GÜLTIGKEITS- UND TRANSAKTIONSZEIT MIT ZEITINTERVALLEN	27
TABELLE 3-7: KORREKTUREN BEI SIMULTANER VERWENDUNG VON GÜLTIGKEITS- UND TRANSAKTIONSZEIT	29
TABELLE 3-8: GÜLTIGKEITS- UND TRANSAKTIONSZEIT MIT MOMENTAUFNAHMEN	32
TABELLE 5-1: INITIALES DATA WAREHOUSE DES PROTOTYPS	53
TABELLE 5-2: METADATEN FÜR DIE HISTORISIERUNGSKOMPONENTE HISTCOMPADB_1	55