Parsen und Transformieren von SQL-MDi-Abfragen in Föderierten Data Warehouse-Systemen

Diplomarbeit
zur Erlangung des akademischen Grades Mag.rer.soc.oec.
im Diplomstudium Wirtschaftsinformatik

Eingereicht an der Johannes Kepler Universität Linz
Institut für Wirschaftsinformatik
Data & Knowledge Engineering

Eingereicht von:
Wolfgang Brunneder

Begutachter:
o. Univ.-Prof. Dr. Michael Schrefl

Betreuender Assistent:
Mag. Stefan Berger

Linz, April 2008

Eidesstattliche Erklärung

Ich erkläre an Eides statt, dass ich die vorliegende Diplomarbeit mit dem Titel „Parsen und Transformieren von SQL-MDi-Abfragen in Föderierten Data Warehouse-Systemen“ selbständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt bzw. die wörtlich oder sinngemäß entnommenen Stellen als solche kenntlich gemacht habe.

Linz, April 2008 Wolfgang Brunneder

Danksagung

An dieser Stelle möchte ich mich bei allen bedanken, die mich beim Verfassen dieser Diplomarbeit unterstützt haben.

Ich bedanke mich bei meinem Betreuer Mag. Stefan Berger, der mir mit seiner konstruktiven Kritik geholfen hat, den Aufbau und Inhalt dieser Diplomarbeit auszuarbeiten. Weiters bedanke ich mich bei Herrn o. Univ.-Prof. Dr. Michael Schrefl, aufgrund dessen Feedback aus dem Diplomarbeitsseminar ich die Qualität der Arbeit noch wesentlich verbessern konnte.

Darüber hinaus bedanke ich mich bei meiner Familie und meinen Freunden, welche mich immer wieder neu motivierten und von etwaigen Zweifeln befreiten. Meiner Freundin Katharina danke ich für ihre liebevolle Unterstützung in der Endphase meiner Diplomarbeit, wodurch ich deren Fertigstellung nochmals beschleunigen konnte. Abschließend gilt mein besonderer Dank meinen Eltern, welche mir mein Studium überhaupt erst ermöglichten.

Wolfgang Brunneder

Zusammenfassung

Unternehmenskooperationen und -zusammenschlüsse können Data Warehouse-übergreifende Analysen zur Leistungsbeurteilung erfordern, wofür sich die Erstellung eines Föderierten Data Warehouse-Systems anbietet. Ein Föderiertes Data Warehouse-System bietet transparenten Zugriff auf die integrierten Komponentensysteme mit Hilfe eines globalen Schemas bzw. eines globalen Würfels. Um Transparenz zu gewährleisten, ist es erforderlich den Benutzer von den zwischen den zu integrierenden Schemata bzw. Würfeln bestehenden Heterogenitäten abzuschirmen. Für diesen Zweck wurde die multidimensionale Abfragesprache SQL-MDi entwickelt, welche die Auflösung der Heterogenitäten ermöglicht.

Ziel dieser Diplomarbeit ist die Entwicklung einer Parser-Komponente, welche eine Teil-Komponente eines Abfrage-Werkzeugs zur Generierung kompatibler Schemata bzw. Würfel in einem Föderierten Data Warehouse-System darstellt. Die Parser-Komponente hat die Aufgabe, eine in SQL-MDi formulierte Abfrage zu analysieren, d.h. die Korrektheit der Syntax und der enthaltenen Metadaten zu prüfen sowie logische Konflikte zu erkennen und zu behandeln. Der vom Parser zu generierende Output ist eine Baumstruktur, welche die Operatoren der SQL-MDi zugrunde liegenden Algebra beinhaltet. Die Baumstruktur dient als Ausführungsplan für eine Prozessor-Komponente, welche auf Basis der Algebra-Operatoren Sub-Anweisungen für die Komponentensysteme erzeugt und ein integriertes Schema bzw. einen integrierten Würfel berechnet. Durch die Berücksichtigung von logischen Optimierungsmöglichkeiten bei der Erzeugung der Baumstruktur, wird die Prozessor-Komponente bei der effizienten Generierung und Verarbeitung der Sub-Anweisungen unterstützt.

Abstract

Business cooperations or mergers may require cross-Data Warehouse analysis to assess business performance. A Federated Data Warehouse System is an appropriate instrument for that purpose by providing transparent access to the integrated component-systems with a global scheme or global cube, respectively. In order to ensure transparency it is necessary to hide existing hetergeneities between the component-schemes from the user. Thus, the multidimenisonal query language SQL-MDi was developed in order to resolve the heterogeneities.

The goal of this thesis is the development of a parser-component as a part of a query tool to generate compatible component-schemes or cubes, respectively, in a Federated Data Warehouse System. One challenge for the parser-component is to analyse a SQL-MDi query by checking the validity of the syntax and contained metadata and by recognizing and handling logical conflicts. The ouput of the parser-component is a tree-structure which contains the operators of the algebra on which SQL-MDi is based on. This tree-structure will be used as an execution plan for a processor-component, which creates statements for the component-systems according to the algebra-operators and calculates an integrated scheme or cube, respectively. Considering logical optimizations during the contruction of the tree-structure enables the processor-component to efficiently create and process the statements for the component-systems.

Abkürzungsverzeichnis

API	Application programming interface
AST	Abstract Syntax Tree (Syntaxbaum)
CWM	Common Warehouse Metamodel
DA	Dimension-Algebra
DBMS	Datenbankmanagementsystem
DBS	Datenbanksystem(e)
DDBMS	Distributed database management system (Verteiltes Datenbankmanagementsystem)
DEA	Deterministischer, endlicher Automat
DFM	Dimensional Fact Model
DW	Data Warehouse/Data Warehousing
DWS	Data Warehouse-System(e)
EBNF	Erweiterter Backus-Naur-Formalismus
ETL	Extraction, Transformation, Loading
FA	Fact-Algebra
FDBS	Föderiertes Datenbanksystem
FDWS	Föderiertes Data Warehouse-System
GAV	Global as view
LAV	Local as view

MDBMS	Multidimensionales Datenbankmanagementsystem
MDBS	Multi-Datenbanksystem(e)
MDWS	Multi-Data Warehouse-System(e)
MFUA	Mobilfunkunternehmen A (des Rahmenbeispiels)
MFUB	Mobilfunkunternehmen B (des Rahmenbeispiels)
MOLAP	Multidimensionales OLAP
NEA	Nichtdeterministischer, endlicher Automat
OLAP	Online analytical processing
OLTP	Online transactional processing
RA	Relationale Algebra
RDBMS	Relationales Datenbankmanagementsystem
ROLAP	Relationales OLAP
UML	Unified Modeling Language

DA/FA-Operatoren

ζ	Rename (DA und FA)
δ	Change (DA)
γ	Convert (DA und FA)
α	Delete level (DA)
Ω	Override rollup (DA)
σ	Select (FA)
π	Project (FA)
λ	Delete measure (FA)
ξ	Pivot (split measure attribute) (FA)
χ	Pivot (merge measure attributes) (FA)
ϵ	Enrich dimensions (FA)
μ	Merge facts (FA)

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Beispielverzeichnis

Definitionen

Kapitel 1

Einleitung

Data Warehouses werden häufig eingesetzt um entscheidungsunterstützende Informationen für die Unternehmensführung bereitzustellen. Im Unterschied zu den operativ eingesetzten, transaktionalen Datenbanken enthalten Data Warehouses umfangreiche historische, zusammengefasste und konsolidierte Datenbestände aus möglicherweise verschiedenen Datenquellen. Diese Datenbestände sind multidimensional in so genannten Würfeln organisiert, auf welchen Analysen durchgeführt werden können. [BG04]

Unternehmenskooperationen und -zusammenschlüsse können Data Warehouse-übergreifende Analysen erfordern um beispielsweise die Wirtschaftlichkeit von unternehmensübergreifenden Geschäftsprozessen beurteilen zu können. Die daraus resultierende Notwendigkeit zur Integration autonomer, unabhängig entwickelter Data Warehouses kann mittels Schaffung eines Verteilten Data Warehouse-Systems oder einer Föderation aus mehreren unabhängigen Data Warehouses (= Föderiertes Data Warehouse-System) erfüllt werden. [BS06a] Da die Implementierung eines Verteilten Data Warehouse-Systems die Ablöse der existierenden, unabhängigen Data Warehouses erfordert und eine kosten-, arbeits- und zeitintensive Aufgabe [AL98] darstellt, ist diese „radikale“ Integrationsvariante in der Praxis oft nicht umsetzbar.

Das im folgenden Abschnitt skizzierte Architekturmodell eines Föderierten Data-Warehouse-Systems bedient sich eines Abfrage-Werkzeugs um unabhängige Data Warehouses zu integrieren. Das Abfrage-Werkzeug verarbeitet Anweisungen einer für den Zweck der Integration von Data Warehouses geschaffenen Abfragesprache. Wichtige Teilaufgaben im Zuge der Verarbeitung sind die Analyse der Korrektheit der Anweisungen sowie die Erzeugung einer intermediären Datenstruktur. Diese Teilaufgaben werden von einem so genannten Parser wahrgenommen, dessen Entwicklung Gegenstand dieser Diplomarbeit ist.

Abschnitt 1.2 behandelt die Themenstellung und die Ziele dieser Diplomarbeit. Das in dieser Arbeit verwendete Rahmenbeispiel, welches der Veranschaulichung der behandelten theoretischen Konzepte dient, wird im Abschnitt 1.3 vorgestellt. Der abschließende Abschnitt 1.4 präsentiert den weiteren Aufbau dieser Diplomarbeit.

1.1 Föderiertes Data Warehouse-System

Ein Föderiertes Data Warehouse-System gewährleistet transparenten Zugriff auf die integrierten Komponentensysteme für die Benutzer [BS08], d.h. das föderierte System wird vom Benutzer wie ein einzelnes Data Warehouse wahrgenommen. Um dies zu erreichen, ist es erforderlich ein globales Schema (einen globalen Würfel) zu erstellen, welches (welcher) die Komponentenschemata (Komponenten-Würfel) integriert. Diese Integrationsaufgabe wird durch zwischen den unabhängigen Schemata und deren Instanzen bestehende Heterogenitäten erschwert. Häufig vorkommende Heterogenitäten bestehen beispielsweise in Namenskonflikten, d.h. wenn semantisch identische Schemaattribute unterschiedliche Namen haben (Synonyme). Abbildung 1.1 zeigt ein vereinfachtes Architekturmodell eines Föderierten Data Warehouse-Systems, welches Transparenz für die Benutzer sicherstellt, indem mit einem Abfrage-Werkzeug ein integrierter, globaler Würfel generiert wird.

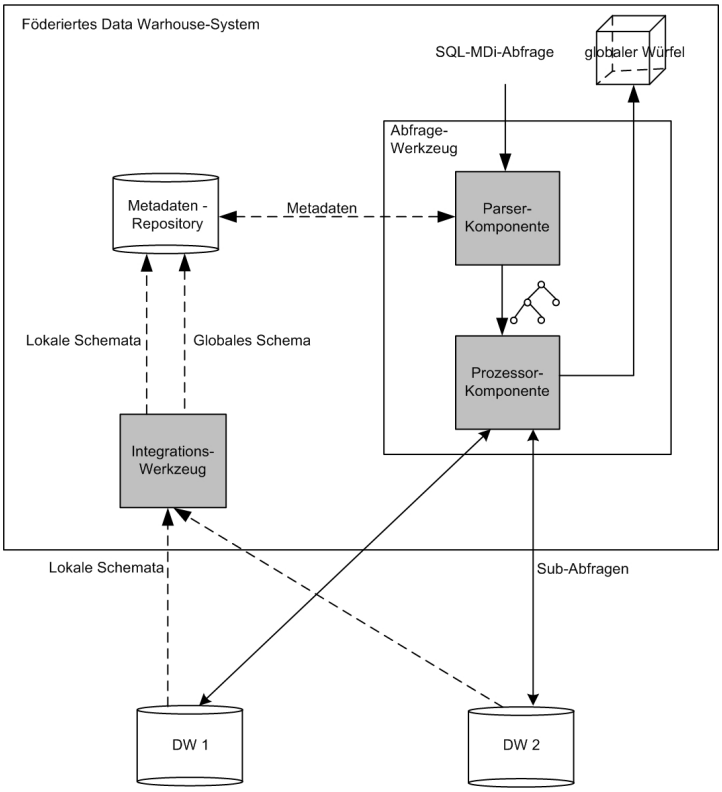


Abbildung 1.1: Föderiertes Data Warehouse-System (vereinfachte Darstellung)

Die Erzeugung des globalen Würfels erfolgt mit der multidimensionalen Abfragesprache SQL-MDi [BS06a] [BS08], welche für den Einsatz in einem Föderierten Data Warehouse-System konzipiert wurde. SQL-MDi unterstützt die Erzeugung eines globalen, instanziierten Würfels, indem Mappings für die Komponentenschemata und -instanzen zur Auflösung von bestehenden Heterogenitäten definiert werden.

Das Föderierte Data Warehouse-System der Abbildung 1.1 umfasst drei Software-Komponenten (grau hinterlegt):

- **Parser-Komponente** (Teil des Abfrage-Werkzeugs): Aufgabe des Parser ist die Analyse der SQL-MDi-Abfrage und die Generierung einer intermediären Datenstruktur, welche Operatoren der SQL-MDi zugrunde liegenden Algebra enthält.
- **Prozessor-Komponente** (Teil des Abfrage-Werkzeugs): Aufgabe des Prozessors ist die Verarbeitung der vom Parser erzeugten Datenstruktur. Dabei werden Sub-Anweisungen für die Komponentensysteme generiert, wodurch der integrierte Würfel berechnet wird.
- **Integrations-Werkzeug**: Das Integrations-Werkzeug unterstützt die Definition eines globalen Data-Warehouse-Schemas aus den Komponenten-Schemata.

1.2 Themenstellung, Zielsetzung und Abgrenzung der Arbeit

Die Themenstellung dieser Diplomarbeit ist die Entwicklung eines Parser für die multidimensionale Abfragesprache SQL-MDi, als Prototyp für den Einsatz in dem im Abschnitt 1.1 vorgestellten Föderierten Data Warehouse-System.

Aufgabe des Parser ist die Analyse einer SQL-MDi-Abfrage und die Erzeugung einer intermediären Datenstruktur in Form eines Baumes, welche Operatoren der SQL-MDi zugrunde liegenden Algebra enthält. Die Analyse einer SQL-MDi-Abfrage umfasst folgende Teilaufgaben:

- Eine SQL-MDi-Abfrage wird auf die Korrektheit der Syntax geprüft. Werden Syntaxfehler erkannt, wird eine aussagekräftige Fehlermeldung erzeugt.
- Eine SQL-MDi-Abfrage wird auf die Korrektheit der enthaltenen Metadaten, d.h. der Schemainformationen, geprüft. Für diesen Zweck steht ein Metadaten-Repository (vgl. Abbildung 1.1) zur Verfügung. Können Metadaten nicht mit dem Repository verifiziert werden, wird eine aussagekräftige Fehlermeldung erzeugt.
- Eine SQL-MDi-Abfrage wird auf logische Konflikte geprüft, welche auf ungültige Kombinationen von Algebra-Operatoren in der Baumstruktur zurückzuführen sind. Werden solche Konflikte erkannt, wird wiederum eine aussagekräftige Fehlermeldung erzeugt.

Die erzeugte Baumstruktur hat folgende Anforderungen zu erfüllen:

- Die Baumstruktur dient als Ausführungsplan für die verarbeitende Prozessor-Komponente. Diese muss aus der Baumstruktur ein integriertes Ergebnis, d.h. den globalen, instanziierten Würfel, erzeugen können, ohne Umstrukturierungen vorzunehmen.
- Um eine effiziente Verarbeitung der Baumstruktur sicherzustellen, erfolgt eine Optimierung auf logischer Ebene, wodurch jene Algebra-Operatoren, welche die Ergebnismenge verkleinern, zu Beginn verarbeitet werden.

In der Disziplin des Übersetzerbaus werden die Aufgaben eines Übersetzers durch unterschiedliche Übersetzungsphasen strukturiert (vgl. Abschnitt 6.2.1). Ein Parser ist für die Übersetzungsphase „Syntaxanalyse“ verantwortlich, in welcher die Korrektheit der Syntax einer (Programmier-)sprache zu prüfen ist. Die Unterstützung der Generierung einer Baumstruktur sowie deren Optimierung ist den Übersetzungsphasen „Semantikverarbeitung“ bzw. „Optimierung“ zugeordnet. Die zusätzliche Implementierung dieser beiden Übersetzungsphasen in der entwickelten Parser-Komponente für SQL-MDi stellt eine Erweiterung des „klassischen“ Aufgabenbereichs eines Parser dar.

Eine ausführlichere Erläuterung der an den Parser gestellten funktionalen und nicht-funktionalen Anforderungen folgt im Abschnitt 7.1.

Neben diesem praktischen Teil wird auch die für die Themenstellung relevante wissenschaftliche Literatur aufgearbeitet, insbesondere zu den Themen Data Warehousing, Integration von Datenbank- und Data Warehouse-Systemen und Übersetzerbau. Im Zuge der Aufarbeitung der wissenschaftlichen Literatur wurden für die Schaffung eines Föderierten Data Warehouse-Systems und für die Entwicklung des Parser in Frage kommende Lösungsansätze analysiert. Während die Behandlung des Themas „Data Warehousing“ (vgl. Kapitel 2) primär der Schaffung eines einheitlichen Begriffsverständnisses und der Einführung der grundlegenden Konzepte dient, wurden zu den Themen „Integration von Datenbank- und Data Warehouse-Systemen“ und „Übersetzerbau“ konkrete Lösungsansätze identifiziert:

- Die Problematik der Integration lässt sich auf Heterogenitäten zurückführen, welche zwischen den zu integrierenden Systemen bestehen. Im Zuge der Aufarbeitung der theoretischen Literatur wurde daher versucht, mögliche Heterogenitäten zwischen unabhängigen Data Warehouses zu kategorisieren und anhand des Rahmenbeispiels zu veranschaulichen. (vgl. Kapitel 3)
- Für die Integration von Datenbanksystemen wurden diverse Architekturmodelle entwickelt, welche zum Teil auch, entsprechend adaptiert, für die Integration von Data Warehouse-Systemen herangezogen werden können. In diesem Zusammenhang sind vor allem Föderierte Datenbanksysteme interessant, da solche Transparenz für die Benutzer gewährleisten, indem von bestehenden Heterogenitäten abstrahiert wird. Auf Basis eines Architekturmodells für Föderierte Datenbanksysteme wurde ein Modell für Föderierte Data Warehouse-Systeme entwickelt. (vgl. Kapitel 3)
- Für die Integration von Datenbanksystemen wurden diverse Abfragesprachen entwickelt, von denen manche auch für die Integration von Data Warehouse-Systemen in Frage

kommen. Dass diese Abfragesprachen dennoch für diesen Zweck wenig geeignet sind, wird anhand des Rahmenbeispiels demonstriert. (vgl. Kapitel 4)

- Für den für die Integration von Data Warehouse-Systemen neu entwickelten Lösungsansatz, die multidimensionale Abfragesprache SQL-MDi, wird anhand des Rahmenbeispiels gezeigt, dass die Integration unabhängiger Data Warehouses umfassend unterstützt wird. Darüber hinaus wird die SQL-MDi zugrunde liegende Algebra erläutert, aus deren Operatoren die Parser-Komponente eine Baumstruktur erzeugt. In diesem Zusammenhang wird die Komposition der Algebra-Operatoren, d.h. die logische Struktur des Operator-Baumes sowie dessen Optimierung, behandelt. (vgl. Kapitel 5)
- Ein Parser kann als Teilkomponente eines Übersetzers betrachtet werden. Ein Übersetzer erfüllt mehrere Teilaufgaben, welche durch Übersetzungsphasen repräsentiert werden. Daher wurden Lösungsansätze für die Realisierung der für die Themenstellung relevanten Übersetzungsphasen aufgearbeitet und bei der Entwicklung der Parser-Komponente berücksichtigt. (vgl. Kapitel 6 und 7)

Die dieser Diplomarbeit zugrunde liegende, vereinfachende Annahme ist, dass die Komponentensysteme des Föderierten Data Warehouse-Systems ein relationales Star-Schema als logisches Modell implementieren (vgl. Abschnitt 2.4.2.1). Auf Basis dieser Annahme war es möglich die für die Themenstellung relevante Theorie stärker einzuschränken bzw. den Aufwand für die Evaluierung der entwickelten Parser-Komponente zu verringern.

1.3 Rahmenbeispiel

Das in diesem Abschnitt erläuterte Rahmenbeispiel dient der Veranschaulichung der dieser Diplomarbeit zugrunde liegenden theoretischen Konzepte, insbesondere der auf dem Gebiet der Integration von Data Warehouse-Systemen geleisteten Forschungsarbeit.

Das Rahmenbeispiel bezieht sich auf einen fiktiven Mobilfunkmarkt eines europäischen Landes. Wir nehmen an, dass ein Mobilfunkunternehmen (MFUA) im Zuge eines Marktkonsolidierungsprozesses ein anderes, im selben Markt tätiges Mobilfunkunternehmen (MFUB) akquiriert. Weiters sei angenommen, dass MFUB ein Tochterunternehmen eines US-amerikanischen Unternehmens darstellt. Beide Mobilfunkunternehmen betreiben ein zentrales Data Warehouse um die Unternehmensführung mit entscheidungsunterstützenden Informationen zu versorgen. Aufgrund der Akquisition weitet sich die Nachfrage nach entscheidungsunterstützenden Informationen der Unternehmensführung von MFUA auf das neue Tochterunternehmen MFUB aus. Um die erweiterte Informationsnachfrage zu befriedigen, ist es erforderlich die beiden unabhängigen, zentralen Data Warehouses zu integrieren. Da die Erstellung und der Betrieb eines Verteilten Data Warehouses für den Zweck der Integration ein zeit- und ressourcenintensiver Prozess ist [AL98], soll das geforderte Informationsangebot mit einem Föderierten Data Warehouse-System zur Verfügung gestellt werden.

Da MFUA und MFUB in derselben Branche tätig sind, ist anzunehmen, dass von der Unternehmensführung gleiche oder ähnliche Informationen nachgefragt werden. Für das Rahmen-

beispiel werden folgende, der Umsatzanalyse dienende Data Warehouse-Schemata für MFUA und MFUB herangezogen:

- Das Data Warehouse-Schema des MFUA speichert Daten zu Mobilfunk-Umsätzen, nämlich die Kenngrößen „Gesprächsdauer (in Min.)“, „Umsatz_Telefonie“ und „Umsatz_Sonstiges“, gegliedert nach den Dimensionen „Datum“, „Kunde“ und „Mobilnetz“. Die Dimension „Datum“ ermöglicht die Analyse für Kalenderdatum, Monat oder Jahr; die Dimension Kunde für einzelne Kunden oder Mobilfunktarife; die Dimension Mobilnetz für gewählte Mobilnetze. Abbildung 1.2 visualisiert dieses Data Warehouse-Schema in Form eines drei-dimensionalen Würfels.
- Das Data Warehouse-Schema des MFUB speichert Daten zu Mobilfunk-Umsätzen, nämlich die Kenngrößen „Gesprächsdauer (in Min.)“ und „Umsatz“, gegliedert nach den Dimensionen „Datum“, „Kunde“, „Mobilnetz“, „Werbeaktion“ und „Umsatzkategorie“, welche der Identifizierung der Umsatz-Art, nämlich Telefonie-Umsatz oder sonstiger Umsatz, dient. Die Dimension „Datum“ ermöglicht die Analyse für Kalenderdatum, Monat, Quartal oder Jahr; die Dimension „Kunde“ für Kunde und Mobilfunktarif; die Dimension „Mobilnetz“ für gewählte Mobilnetze; die Dimension „Werbeaktion“ für einem Umsatz zugeordnete Werbeaktionen; die Dimension „Umsatzkategorie“ für einem Umsatz zugeordnete Umsatzkategorien. Abbildung 1.2 visualisiert dieses Data Warehouse-Schema in Form eines fünf-dimensionalen Würfels (aufgrund der Schwierigkeit fünf Dimensionen graphisch darzustellen, wurde ein drei-dimensionaler Würfel verwendet).

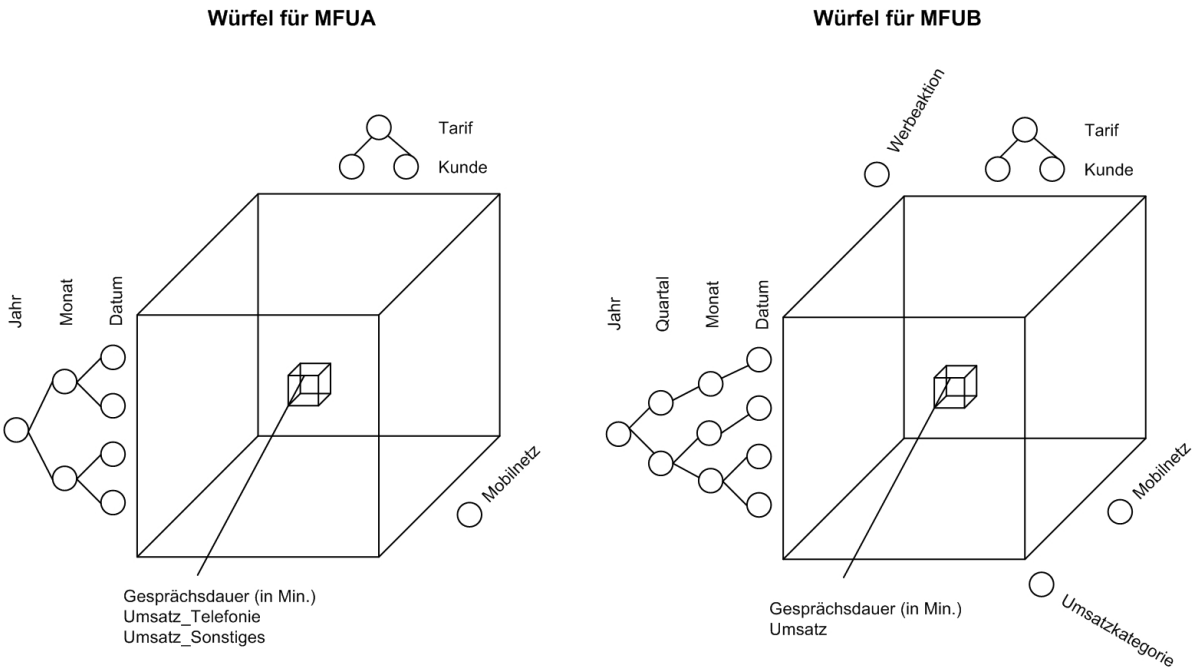


Abbildung 1.2: Würfel für MFUA und MFUB

Die Konstrukte für die Modellierung von Data Warehouses, z.B. Würfel und Dimensionen, und der Entwurfsprozess für Data Warehouse-Schemata werden in Kapitel 2 beschrieben.

Anhand dieses Rahmenbeispiels werden im Teil I dieser Diplomarbeit „Data Warehousing und integrierte Data Warehouse-Systeme“ die Grundlagen des Data Warehousing, insbesondere der Entwurfsprozess für ein Data Warehouse, veranschaulicht sowie mögliche Konflikte, welche die Integration von Data Warehouses erschweren, erläutert. Im Teil II „Sprachen für die Integration von Data Warehouse-Systemen“ werden ausgewählte Abfragesprachen, inklusive der multidimensionalen Abfragesprache für die Integration von Data Warehouses, SQL-MDi, dahingehend untersucht, ob sie zur Auflösung der im Rahmenbeispiel bestehenden Heterogenitäten geeignet sind und ggf. Defizite aufgezeigt.

1.4 Aufbau der Arbeit

Der Aufbau der Diplomarbeit gliedert sich in drei Teile:

Der Teil I „Data Warehousing und integrierte Data Warehouse-Systeme“ behandelt Data Warehouses und Systeme, welche die Integration von unabhängigen Data Warehouses unterstützen. Im Kapitel 2 werden die theoretischen Grundlagen des Data Warehousing erläutert, insbesondere das für Data Warehouses charakteristische multidimensionale Datenmodell und eine Data Warehouse-Design-Methodologie. Kapitel 3 behandelt Ansätze für die Integration von Datenbanken und Data Warehouses, wobei der Schwerpunkt auf der Verdeutlichung der Problematik der Heterogenität bei der Integration von Data Warehouse-Systemen liegt. In diesem Zusammenhang wird als Lösungsansatz ein Architekturmodell eines Föderierten Data Warehouse-Systems präsentiert.

Der Teil II „Sprachen für die Integration von Data Warehouse-Systemen“ behandelt die systeminterne Sicht, d.h. wie Integration erreicht werden kann. Zu diesem Zweck werden im Kapitel 4 diverse Abfragesprachen für die Integration von Datenbank- und Data Warehouse-Systemen präsentiert. Jene Abfragesprachen, welche für die Data Warehouse-Integration am geeignetsten erscheinen, werden ausführlicher behandelt. In diesem Zusammenhang wird demonstriert, ob bzw. wie die bei der Integration auftretenden Heterogenitäten eliminiert werden können. Die im Abschnitt 1.1 erwähnte multidimensionale Abfragesprache SQL-MDi wird mitsamt ihrer zugrunde liegenden, formalen Algebra im Kapitel 5 erläutert. Für diese Abfragesprache wird gezeigt, dass sie im Stande ist, die bei der Integration auftretenden Heterogenitäten mit einer einzigen Abfrage aufzulösen.

Der Teil III „Parser-Implementierung“ umfasst die theoretischen Grundlagen für die Entwicklung einer Parser-Komponente sowie eine Beschreibung des entwickelten Parser für SQL-MDi. Kapitel 6 behandelt das grundlegende Thema der formalen Sprachen und Automaten sowie die Grundlagen des Übersetzerbaus, wovon die Entwicklung eines Parser ein Teilgebiet darstellt. Das abschließende Kapitel 7 dient der Beschreibung der an den entwickelten Parser gestellten Anforderungen sowie deren Erfüllung im Zuge der Beschreibung der Systemarchitektur und -implementierung. Es wird veranschaulicht, wie die Berücksichtigung der Konzepte des Übersetzerbaus die Gestaltung der Systemarchitektur beeinflusste. Die Beschreibung der Systemimplementierung demonstriert, wie der Parser die im Abschnitt 1.2 erläuterten Aufgaben

erfüllt, d.h. wie eine SQL-MDi-Abfrage analysiert und wie eine Baumstruktur, bestehend aus Algebra-Operatoren, erzeugt und optimiert wird.

Am Ende der Arbeit wird in Kapitel 8 zunächst erläutert, wie der entwickelte Parser evaluiert wurde, d.h. wie sichergestellt wurde, dass die vom Parser generierte, intermediäre Datenstruktur durch eine Prozessor-Komponente verarbeitet und ein globaler Würfel berechnet werden kann. Abschließend folgt ein Resumee sowie ein Überblick über noch offene Fragen.

Teil I

Data Warehousing und integrierte Data Warehouse-Systeme

Kapitel 2

Data Warehousing

Im Zusammenhang mit Systemen zur Entscheidungsunterstützung sind die wohl am häufigsten genannten Begriffe „Data Warehousing“ bzw. „Data Warehouse“ und „On-line analytical processing (OLAP)“. Um eine missbräuchliche Verwendung der Begriffe zu vermeiden, werden diese im folgenden Abschnitt untereinander sowie zu den traditionellen, transaktionalen Datenbanken abgegrenzt. Ein typisches Modell einer Data Warehousing-Architektur mit ihren wichtigsten Komponenten wird im Abschnitt 2.2 vorgestellt. Die Konstrukte des für Data Warehouses charakteristischen, multidimensionalen Datenmodells werden im Abschnitt 2.3 erläutert. Im Abschnitt 2.4 wird der Entwurfsprozess für Data Warehouses mit Hilfe einer Design-Methodologie, mit den Phasen „konzeptuelles Design“, „logisches Design“ und „physisches Design“, behandelt. Der letzte Abschnitt 2.5 stellt eine Zusammenfassung des Kapitels dar.

2.1 Einordnung und Abgrenzung

In [Inm05] wird ein Data Warehouse als „subject-oriented, integrated, non-volatile, and time variant collection of data in support of management's decisions“ definiert. Aus dieser Definition lassen sich vier Eigenschaften ableiten, anhand derer ein Data Warehouse charakterisiert werden kann: [BG04]

- **Fachorientierung** (engl. subject orientation): Ein Data Warehouse modelliert einen spezifischen Anwendungsbereich.
- **Integrierte Datenbasis** (engl. integration): Es wird der integrierte Datenbestand mehrerer Datenbanken verarbeitet.
- **Nicht flüchtige Datenbasis** (engl. non-volatile): Daten im Data Warehouse werden nicht mehr entfernt oder geändert.
- **Historische Daten** (engl. time variance): Bei der Datenverarbeitung finden Vergleiche über die Zeit statt.

Der Begriff „Data Warehousing“ bezeichnet den Data Warehouse-Prozess, also den dynamischen Vorgang, angefangen vom Prozess der Datenintegration bis hin zur Analyse der integrierten Datenbestände durch die Benutzer. [BG04]

Fälschlicherweise wird der Begriff OLAP oft mit Data Warehousing bzw. Data Warehouses gleichgesetzt. OLAP ist jedoch vielmehr ein Teilbereich des Data Warehousing und baut auf Data Warehouses auf, indem es dynamische, flexible und interaktive Analysen auf deren Datenbeständen ermöglicht. Bezogen auf das Rahmenbeispiel, beschäftigt sich OLAP mit Fragestellungen wie z.B.: „In welchem Jahr konnte mit welchem Mobilfunktarif der größte Umsatz erwirtschaftet werden?“. [BG04]

Das mit Datenbanken im Zusammenhang stehende „On-line transactional processing (OLTP)“ dient der Automatisierung von täglich auftretenden Geschäftsfällen (z.B. Auftragseingänge oder Banktransaktionen). Die Automatisierungsaufgaben werden mit Lese- und Schreib-Transaktionen für detaillierte und aktuelle Daten abgewickelt. Data Warehouses zielen hingegen auf Entscheidungsunterstützung ab. Sie beinhalten konsolidierte Datenbestände, welche die Basis für OLAP-Analysen bilden. Im Gegensatz zum OLTP müssen im Rahmen des Data Warehousing keine laufenden Schreib-Transaktionen gehandhabt werden; OLAP-Analysen sind mit komplexen Lese-Transaktionen verbunden. Da die aus mehreren operativen Datenquellen konsolidierten Datenbestände eines Data Warehouse in der Regel deutlich größer sind als in operativen Datenbanken, werden besondere Anforderungen an die Effizienz der Verarbeitung von Abfragen gestellt. [CD97]

In den weiteren Ausführungen wird für „Data Warehouse“ in zusammengesetzten Begriffen meist die Abkürzung „DW“ verwendet.

2.2 Data Warehousing-Architektur

In diesem Abschnitt wird anhand der Abbildung 2.1 eine typische Data Warehousing-Architektur vorgestellt:

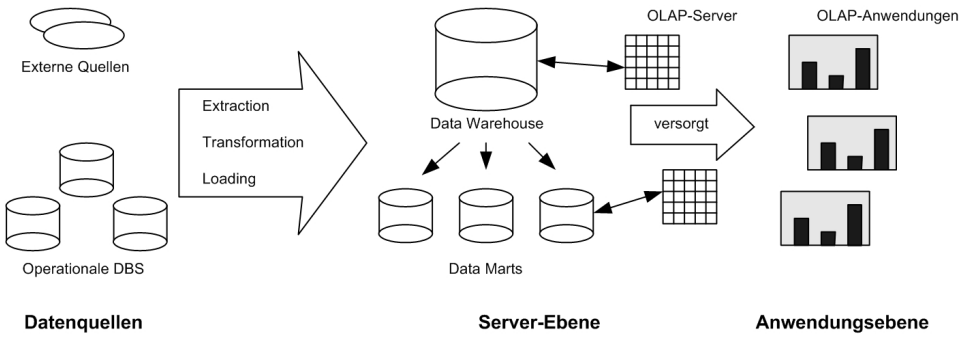


Abbildung 2.1: Data Warehousing-Architektur (vgl. [CD97])

Ein Data Warehouse integriert Datenbestände aus operativen Datenquellen, oft Datenbanken, wofür so genannte ETL(Extraction, Transformation, Loading)-Werkzeuge existieren. Die ETL-Prozesse zur „Befüllung“ des Data Warehouse sind die einzigen Schreib-Operationen, welche im Data Warehousing auftreten. [Inm05]

Für die Entscheidungsunterstützung auf Abteilungsebene werden neben dem Data Warehouse oft so genannte Data Marts erstellt, welche nur die für die jeweilige Abteilung relevanten Daten integrieren. Während im Data Warehouse konsolidierte Detaildaten in der Regel normalisiert gespeichert werden, enthalten Data Marts davon abgeleitete zusammengefasste, denormalisierte Daten, welche für die jeweilige Abteilung von Bedeutung sind. Ein wesentliches Charakteristikum für Data Warehouses und Data Marts ist die Verwendung eines multidimensionalen Datenmodells, welches der Multidimensionalität von OLAP-Analysen gerecht wird (z.B. Analyse der Umsätze anhand der Dimensionen „Zeit“ und „Mobilfunktarif“). Für die Implementierung eines multidimensionalen Datenmodells sind zwei Speichermodelle vorherrschend: die relationale Speicherung (ROLAP - relationales OLAP) und die multidimensionale Speicherung (MOLAP - multidimensionales OLAP) (vgl. Abschnitt 2.4.2). Die konsolidierten Datenbestände des Data Warehouse werden in der Regel relational, die der Data Marts entweder relational oder multidimensional gespeichert. [Inm05]

Die Daten im Data Warehouse und in den Data Marts werden von einem oder mehreren OLAP-Servern gespeichert, verwaltet und in einer multidimensionalen Form, in so genannten Würfeln, für OLAP-Anwendungen aufbereitet. Abhängig vom bevorzugten Speichermodell für das Data Warehouse bzw. die Data Marts, kann die Data Warehousing-Architektur sowohl relationale als auch multidimensionale OLAP-Server umfassen. [Inm05]

Wie aus der Abbildung 2.1 und den bisherigen Erläuterungen hervorgeht, ist ein Data Warehouse nur ein Bestandteil der Data Warehousing-Architektur. Alle für die Datenintegration und -analyse notwendigen Komponenten werden unter dem Begriff „Data Warehouse-System“ subsummiert [BG04]. Da das Thema dieser Diplomarbeit die Integration unabhängiger Data Warehouses betrifft, wird in den folgenden Kapiteln des öfteren der Begriff „Data Warehouse-System“ als Synonym für „Data Warehouse“ bzw. „Data Marts“ verwendet.

Dieser Diplomarbeit liegt die Annahme zugrunde, dass die zu integrierenden Komponentensysteme relational modelliert sind (vgl. Kapitel 1). Somit integriert das gegenständliche Föderierte Data Warehouse-System eine Menge Data Warehouses bzw. Data Marts, welche von einem relationalen OLAP-Server verwaltet werden.

Der nächste Abschnitt ist der Erläuterung des multidimensionalen Datenmodells gewidmet.

2.3 Multidimensionales Datenmodell

Die Wichtigkeit eines formal beschriebenen, einheitlichen Datenmodells lässt sich am Beispiel des relationalen Datenmodells [Cod70] verdeutlichen. Das relationale Datenmodell wurde entwickelt, um Datenstrukturen unabhängig von der physischen Implementierung beschreiben zu können. Da dieser Formalismus rasch akzeptiert wurde, konnten Forschungsergebnisse schnell in Produkten realisiert werden. [BG04] Das in diesem Abschnitt informell beschriebene, multidimensionale Datenmodell ist semantisch reichhaltiger als das relationale Datenmodell, welches im Wesentlichen nur Entitäten mit Attributen und Beziehungen zwischen Entitäten kennt [KE04]. Die Bestandteile des multidimensionalen Datenmodells werden gemäß [BG04] und [GMR98] erläutert.

Definition 2.1. Ein *Würfel* besteht aus einer Menge von Zellen bzw. Fakten, welche eine oder mehrere Kenngrößen beinhalten. Eine Zelle bildet dabei den Schnittpunkt der Dimensionen, welche die Achsen des Würfels darstellen und diesen aufspannen. Die Dimensionalität des Würfels ergibt sich aus der Anzahl der ihn aufspannenden Dimensionen. [BG04]

Beispiel 2.1: Im Rahmenbeispiel könnte ein Würfel „Verkauf“ modelliert werden, welcher die Analyse von Umsätzen anhand definierter Dimensionen erlaubt.

Definition 2.2. Eine *Kenngröße* ist in der Regel eine nummerische Kennzahl, welche aus verschiedenen Blickwinkeln (anhand unterschiedlicher Dimensionen) untersucht werden kann. Jede Kenngröße kann als Funktion der sie charakterisierenden Dimensionen verstanden werden. [BG04]

Beispiel 2.2: Im Rahmenbeispiel könnten die Zellen des Würfels „Verkauf“ eine Kenngröße „Umsatz“ beinhalten.

Definition 2.3. Eine *Dimension* stellt eine ausgewählte Entität dar, mit welcher eine Analysesicht eines Anwendungsbereichs definiert wird. Dimensionen stellen die Achsen eines Würfels dar und dienen dessen eindeutiger, orthogonaler Strukturierung. [BG04]

Beispiel 2.3: Im Rahmenbeispiel könnte der Würfel „Verkauf“ anhand der Dimensionen „Datum“, „Kunde“ und „Mobilnetz“ aufgespannt werden.

Definition 2.4. Eine *Klassifikationsstufe* (in [GMR98] als dimensionales Attribut bezeichnet) modelliert eine bestimmte Granularität, mit der eine Dimension beschrieben wird. Klassifikationsstufen dienen somit der hierarchischen Organisation der Dimensionselemente. Dimensionselemente sind die Ausprägungen der Klassifikationsstufe niedrigster bzw. feinster Granularität. Zur Dimensionsinstanz werden neben den Dimensionselementen auch die Ausprägungen von Klassifikationsstufen höherer bzw. gröberer Granularität gezählt. Zur Klassifikationsstufe niedrigster Granularität (= Basis-Klassifikationsstufe) (z.B. Tag) werden die atomaren Werte für die Kenngrößen hinterlegt. Klassifikationsstufen höherer Granularität enthalten aggregierte Werte für die jeweilige Klassifikationsstufe niedrigerer Granularität (z.B. enthält ein Monat einen aggregierten Wert für die ihm zugeordneten Tage). [BG04]

Beispiel 2.4: Im Rahmenbeispiel könnte die Dimension „Datum“ die Menge {Tag, Monat, Jahr} als Klassifikationsstufen umfassen. Die Dimensionselemente der Dimension „Datum“ bilden die einzelnen Tage; die Dimensionsinstanz umfasst zusätzlich die den einzelnen Tagen zugeordneten Monate und Jahre.

Definition 2.5. *Roll-up Beziehungen* (Bezeichnung in Anlehnung an [CT05]) repräsentieren Abhängigkeiten zwischen Klassifikationsstufen unterschiedlicher Granularität. Die Abhängigkeiten stellen hierarchische n:1 Beziehungen dar, d.h. n Dimensionsinstanzen niedrigerer Granularität sind genau 1 Dimensionsinstanz höherer Granularität zugeordnet. [BG04]

Beispiel 2.5: Im Rahmenbeispiel würde eine roll-up Beziehung Tag $\mapsto$ Monat der Dimension „Datum“ ausdrücken, dass jeder Tag genau einem Monat zuordenbar ist, d.h. 01.01.2007 $\mapsto$ 01/2007, 01.06.2007 $\mapsto$ 06/2007, etc.

Definition 2.6. Eine *Hierarchie* (in [BG04] als Klassifikationshierarchie bezeichnet) ist ein balancierter Baum, dessen Knotenmenge aus den Wertebereichen der Klassifikationsstufen, erweitert um eine (Wurzel-)Klassifikationsstufe „all“, besteht und dessen Kanten den bestehenden roll-up Beziehungen entsprechen. Die Klassifikationsstufe „all“ ist die oberste Klassifikationsstufe (gröbster Granularität) einer Hierarchie und stellt eine Aggregation auf einen einzelnen Wert dar. Das Schema der Hierarchie bzw. des Baumes, welcher aus den Wertebereichen der Klassifikationsstufen zuzüglich „all“ besteht, dient der Beschreibung einer Dimension und wird als „Dimensionsschema“ bezeichnet. [BG04]

Beispiel 2.6: Im Rahmenbeispiel könnte die Dimension „Datum“ eine Menge von Klassifikationsstufen {Tag, Monat, Jahr, all} in der Hierarchie Tag $\mapsto$ Monat $\mapsto$ Jahr $\mapsto$ all beinhalten.

Definition 2.7. Ein *nicht-dimensionales Attribut* [GMR98] kann einer Klassifikationsstufe zugeordnet werden und dient deren näherer Beschreibung, d.h. es kann zusätzliche Information zu einer Klassifikationsstufe gespeichert werden. Nicht-dimensionale Attribute haben keinen Einfluss auf die Analyseergebnisse, da sie nicht für die Berechnung von Aggregationen verwendet werden können. [GMR98]

Beispiel 2.7: Im Rahmenbeispiel könnte der Klassifikationsstufe „Monat“ der Dimension „Datum“ ein nicht-dimensionales Attribut „Name“ zugeordnet werden, welches die Bezeichnung des Monats speichert (z.B. Januar für 01/2007, Februar für 02/2007, etc.).

Eine erweiterte, formale Definition des multidimensionalen Datenmodells, welches die formale Grundlage der multidimensionalen Abfragesprache SQL-MDi bildet, erfolgt im Abschnitt 5.3.2.

Entsprechend der Annahme, dass das gegenständliche Föderierte Data Warehouse-System ROLAP-Systeme integriert, wird im folgenden Abschnitt eine Methodik vorgestellt, welche die relationale Umsetzung des multidimensionalen Datenmodells strukturiert.

2.4 Data Warehouse-Design

In diesem Abschnitt wird demonstriert, wie auf Basis des im vorherigen Abschnitt vorgestellten multidimensionalen Datenmodells Schemata bzw. Würfel für Data Warehouses und Data Marts entworfen werden können. Aufgrund der multidimensionalen Datenmodellierung ist im Vergleich zu relationalen Systemen eine adaptierte Design-Methodologie erforderlich. Im Folgenden werden drei Phasen der in [GR98] vorgestellten Design-Methodologie für Data Warehouses erläutert:

- konzeptuelles Design
- logisches Design
- physisches Design

Die drei Phasen, konzeptuelles Design, logisches Design und physisches Design, finden auch beim Entwurf von Datenbanken Verwendung [KE04]. Unabhängig davon, ob die Phasen den Entwurf von Datenbanken oder Data Warehouses strukturieren, werden folgende Ziele verfolgt: Das Ziel der Phase „konzeptuelles Design“ ist die Erstellung eines von Implementierungsaspekten unabhängigen, problemnahen Datenmodells. Im Zuge des logischen Designs wird ein Modell entwickelt, welches die Kluft zwischen konzeptuellem und physischem Modell verringern und möglichst automatisch in letzteres transformiert werden kann. Aufgabe des physischen Designs ist die Abbildung des logischen Modells auf die speziellen Möglichkeiten der Implementierungsplattform, also die Erstellung des physischen Modells.

Durch die Strukturierung des Entwurfsprozesses mittels Phasen kann eine Trennung von Informationsanforderungen und Implementierungsanforderungen erreicht werden. Das konzeptuelle Design ist primär von den Informationsanforderungen getrieben, welche von den Fachabteilungen festgelegt werden; das physische Design primär von den Charakteristika der Implementierungsplattform (z.B. Datenbanksystem), welche den für die Implementierung verantwortlichen Personen bekannt sein müssen. Das logische Design bildet eine Brücke zwischen den beiden Phasen, indem sowohl Informations- als auch Implementierungsaspekte berücksichtigt werden. [KE04]

In den folgenden Abschnitten werden die Inhalte der drei Design-Phasen für das Data Warehousing erläutert.

2.4.1 Konzeptuelles Design

Im Zuge des konzeptuellen Designs wird ein konzeptuelles, von einer konkreten Implementierungsplattform unabhängiges DW-Modell erstellt, welches eine Beschreibung der Struktur eines Data Warehouse auf Schema- und Instanzebene darstellt. Das in [GMR98] vorgestellte „Dimensional Fact Model (DFM)“ formalisiert die Erstellung eines graphischen, konzeptuellen Modells für Data Warehouses und basiert auf dem im Abschnitt 2.3 vorgestellten multidimensionalen Datenmodell. Im Folgenden werden die Grundlagen des DFM auf Schema- und Instanzebene vorgestellt.

2.4.1.1 Schemaebene

Ein nach der DFM-Notation erstelltes, dimensionales Schema besteht aus einer Menge von Faktschemata, welche aus Fakten, Kenngrößen, Dimensionen und Hierarchien bestehen.

- Ein Faktschema entspricht einem Würfel des multidimensionalen Datenmodells. Ein Fakt ist der Mittelpunkt des Interesses im Entscheidungsfindungsprozess, er stellt Daten zu einem betrieblichen Ereignis dar (z.B. zu einem Verkauf). (vgl. Definition 2.1)
- Kenngrößen sind in der Regel numerische Attribute, welche einen Fakt aus verschiedenen Blickwinkeln beschreiben (z.B. kann ein Verkauf durch den Umsatz beschrieben werden). (vgl. Definition 2.2)
- Dimensionen sind Attribute, welche die minimale Granularität festlegen, mit der ein Fakt beschrieben wird (z.B. ist Datum eine Dimension für einen Verkauf). (vgl. Definition 2.3)
- Hierarchien bestehen aus dimensionalen Attributen und bestimmen wie Fakten im Zuge von Analysen aggregiert und ausgewählt werden können. Die Wurzel einer Hierarchie ist verbunden mit dem dimensionalen Attribut feinster Granularität; die weiteren dimensionalen Attribute der Hierarchie definieren gröbere Granularitäten. Darüber hinaus können Dimensionen auch nicht-dimensionale Attribute umfassen. Diese dienen der näheren Beschreibung eines dimensionalen Attributs, können jedoch nicht für Aggregationen verwendet werden. (vgl. Definitionen 2.4 - 2.7)

Abbildung 2.2 zeigt das Faktschema „Verkauf“ des Data Warehouse MFUA:

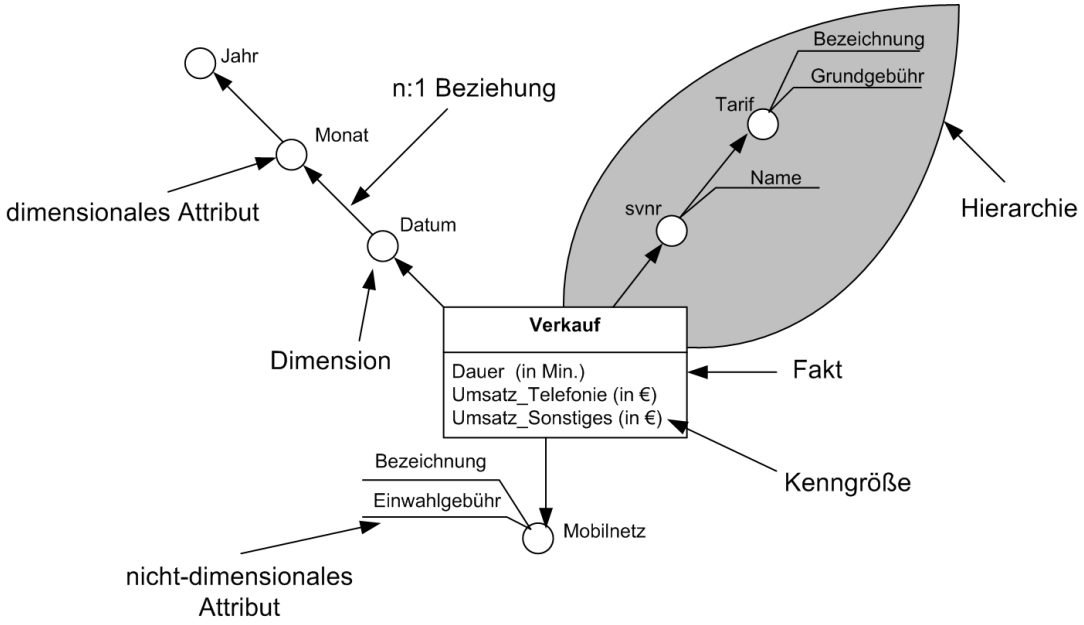


Abbildung 2.2: Fakt Schema: Verkauf (MFUA)

Das Rechteck repräsentiert den Fakt (vgl. Definition 2.1) „Verkauf“ und beinhaltet mehrere Kenngrößen (vgl. Definition 2.2), nämlich „Dauer (in Min.)“, „Umsatz_Telefonie (in €)“ und „Umsatz_Sonstiges (in €)“. Dimensionale Attribute (vgl. Definition 2.4) werden durch Kreise dargestellt. Jedes dimensionale Attribut, das direkt mit dem Fakt verbunden ist, bezeichnet eine Dimension (vgl. Definition 2.3). Demnach ist die Menge der Dimensionen des obigen Faktschemas {Datum, svnr, Mobilnetz}. In diesem Zusammenhang ist anzumerken, dass im multidimensionalen Datenmodell der Name einer Dimension vom Namen des mit einem Fakt verbundenen dimensionalen Attributs abweichen kann, z.B. Dimension „Kunde“ statt „svnr“ [BG04]. Im Folgenden wird daher der Name „Kunde“ für diese Dimension verwendet. Nicht-dimensionale Attribute (vgl. Definition 2.7) werden durch an die dimensionalen Attribute angeschlossene Linien dargestellt, z.B. „Bezeichnung“ für „Tarif“. Subbäume, deren Wurzel in einer Dimension liegt, stellen eine Hierarchie (vgl. Definition 2.6) dar, z.B. die Hierarchie der Dimension „Datum“ ist $\{Datum \rightarrow Monat \rightarrow Jahr \rightarrow all\}$, was auch als Dimensionsschema bezeichnet wird [BG04]. Jede Hierarchie enthält eine Klassifikationsstufe „all“, welche die Menge der Klassifikationsstufen der Hierarchie repräsentiert. Dadurch kann ein einzelner, aggregierter Wert für die entsprechende Dimension gebildet werden, wodurch diese „ausgeblendet“ werden kann. [GCB+97] [BG04]. Ein Pfeil zwischen zwei dimensionalen Attributen einer Hierarchie stellt eine n:1 Beziehung bzw. roll-up Beziehung (vgl. Definition 2.5) dar. Eine roll-up Beziehung ist die Abbildung einer Dimensionsinstanz niedrigerer Granularität auf eine Dimensionsinstanz höherer Granularität.

2.4.1.2 Instanzebene

In einem gegebenen Faktschema f, repräsentiert ein n-Tupel von Werten aus den Domänen der n Dimensionen von f eine Zelle des Würfels, welche eine Informationseinheit in einem Data Warehouse bzw. in einem Data Mart darstellt. „Primäre Faktinstanzen“ sind jene Informationseinheiten, welche genau einen Wert für jede Kenngröße haben. Die Identifikation von primären Faktinstanzen erfolgt mit einer Kombination von Ausprägungen dimensionaler Attribute; anders ausgedrückt: die Ausprägungen der dimensionalen Attribute bilden die Koordinaten für eine bestimmte Kenngröße bzw. Zelle im Würfel. Bezogen auf das Rahmenbeispiel bedeutet das, dass jede primäre Faktinstanz die Verkäufe pro Datum, pro svnr und pro Mobilnetz beschreibt (vgl. Abbildung 2.3). [GMR98]

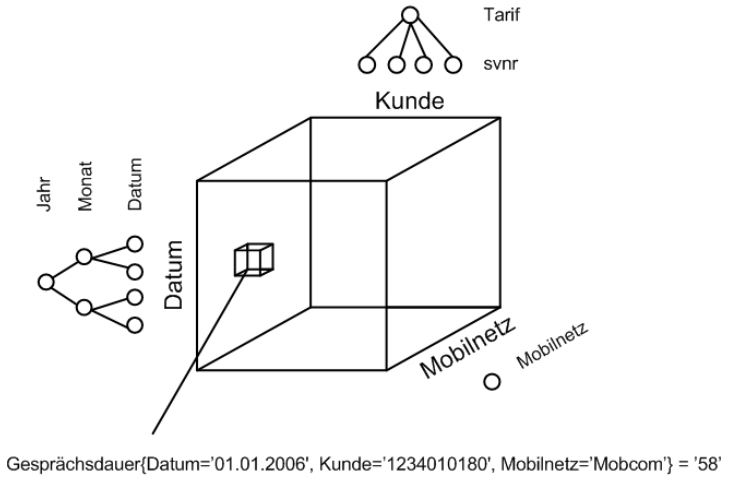


Abbildung 2.3: Primäre Faktinstanz: Verkauf (MFUA)

Da Analysen auf der feinsten Granularitätsstufe für strategische Entscheidungen oft ungeeignet sind, bietet sich an, die primären Faktinstanzen auf verschiedenen Abstraktionsstufen, durch Wahl gröberer Granularitäten in einer oder mehreren Dimensionen, zu aggregieren. Im OLAP-Kontext bezeichnet man dieses Vorgehen mit „roll-up“, welches sich mit „aggregation patterns“ beschreiben lässt:

Beispiel 2.8: Ein mögliches roll-up im Rahmenbeispiel wäre `VERKAUF(Datum.Monat, Kunde.Tarif, Mobilnetz).Umsatz_Telefonie (SUM)` wodurch die Summe der Telefonie-Umsätze pro Monat, Tarif und Mobilnetz berechnet wird.

Auf solche Weise aggregierte, primäre Faktinstanzen nennt man „sekundäre Faktinstanzen“. [GMR98] Weitere klassische OLAP-Operationen sind „drill-down“ (Gegenteil von roll-up), „slice“ und „dice“ (Selektion und Projektion) sowie „pivot“ (Umstrukturieren des Würfels) [CD97].

In diesem Abschnitt wurde das DFM dazu verwendet, das vorgestellte multidimensionale Datenmodell bzw. die DW-Entwurfsphase „konzeptuelles Design“ zu erläutern. Wie bereits

erwähnt, ist ein konzeptuelles Modell (z.B. nach der DFM-Notation) unabhängig von der Implementierung des Data Warehouse. Mit dieser beschäftigen sich die Phasen „logisches Design“ und „physisches Design“, welche nachfolgend behandelt werden.

2.4.2 Logisches Design

Diese Phase umfasst die Entscheidung, wie das in der vorangegangenen Phase entwickelte konzeptuelle Modell auf einer Implementierungsplattform umgesetzt wird. Die Speicherung des Data Warehouse erfolgt in einer Datenbank, welche von einem Datenbankmanagementsystem (DBMS) betrieben wird. Im DW-Umfeld sind zwei Speichermodelle vorherrschend: die relationale Speicherung und die multidimensionale Speicherung. [BG04]

Im relationalen Speichermodell, auch ROLAP (relationales OLAP) genannt, werden die DW-Konstrukte, Fakten und Dimensionen, in einem relationalen Datenbankmodell, d.h. in Relationenschemata, organisiert. Dadurch wird ein multidimensionaler Würfel durch mehrere zweidimensionale Tabellen dargestellt. Relational modellierte Data Warehouses bzw. Data Marts werden von einem relationalen Datenbankmanagementsystem (RDBMS) verwaltet (Relationaler OLAP-Server [CD97]). Im multidimensionalen Speichermodell, auch MOLAP (multidimensionales OLAP) genannt, wird das multidimensionale Datenmodell direkt in multidimensionale Datenstrukturen (meist Arrays) umgesetzt, wofür ein multidimensionales Datenbankmanagementsystem (MDBMS) benötigt wird (Multidimensionaler OLAP-Server [CD97]). [BG04]

In der Praxis hat sich das relationale Speichermodell weitgehend durchgesetzt. Die Gründe dafür liegen zum einen darin, dass relationale Datenbanken weit verbreitet sind und deren Technologie sehr ausgereift ist. Darüber hinaus zeigen relationale Systeme ein besseres Skalierungsverhalten als multidimensionale Systeme, welche jedoch eine effizientere Abfrageverarbeitung in Anwendungen mit geringem Datenumfang gewährleisten. [BG04] Aufgrund der Dominanz des relationalen Speichermodells, wird auf eine Erläuterung des multidimensionalen Ansatzes verzichtet.

Im relationalen Speichermodell gestaltet sich die Abbildung eines Würfels, ohne Berücksichtigung von Dimensionshierarchien, auf ein relationales Schema einfach, da jede Relation bzw. Tabelle als Würfel betrachtet werden kann, indem ein Teil der Attribute als Dimensionen und die restlichen als Kenngrößen interpretiert werden. Ein Tupel der Relation stellt einen Fakt dar und entspricht einer Zelle im Würfel. [BG04] Aufgrund des konzeptuellen Datenmodells des MFUA könnte das logische Modell (Relationenschema, Fakttabelle) der Abbildung 2.4 erstellt werden (ohne Berücksichtigung der Dimensionshierarchien):

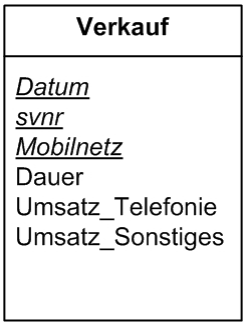


Abbildung 2.4: Fakttabelle: Verkauf (MFUA)

Die Attribute {Datum, svnr, Mobilnetz} stellen die Dimensionen des Würfels dar; die Attribute {Dauer, Umsatz_Telefonie, Umsatz_Sonstiges} die Kenngrößen. Für die Abbildung der Dimensionen samt Hierarchien werden im Folgenden zwei unterschiedliche Ansätze vorgestellt, nämlich das Star- und das Snowflake-Schema.

2.4.2.1 Star-Schema

Im Star-Schema wird für jede Dimension genau eine Dimensionstabelle modelliert, welche mittels Fremdschlüsselbeziehungen mit der Fakttabelle verbunden sind. [BG04] Abbildung 2.5 visualisiert das logische Datenmodell für MFUA als Star-Schema:

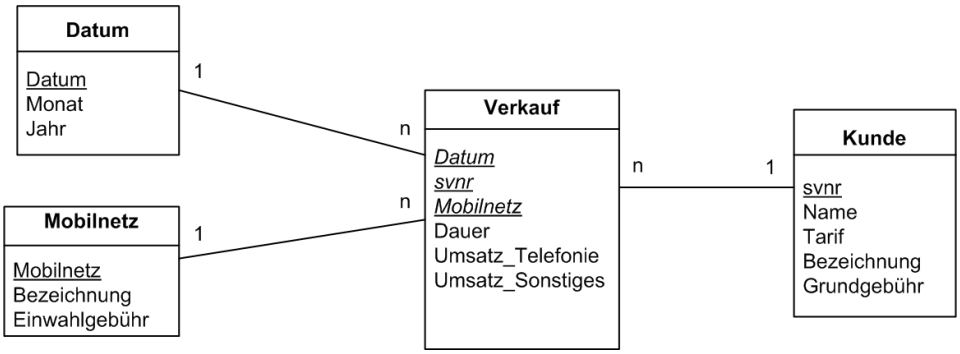


Abbildung 2.5: Star-Schema: Verkauf (MFUA)

Die Dimensionstabellen enthalten die der entsprechenden Dimension zugeordneten Klassifikationsstufen sowie die diesen zugeordneten nicht-dimensionalen Attribute. Die unterstrichenen Felder der Tabellen stellen die Primärschlüssel dar, die kursiv gedruckten, Fremdschlüssel. Man beachte, dass sich der Primärschlüssel der Fakttabelle aus der Menge der Fremdschlüssel der Dimensionen bildet. Die Tupel der Dimensionstabellen stellen Dimensionsinstanzen dar. [BG04]

In der Praxis wird für den Primärschlüssel der Dimensionstabellen oft eine laufende Nummer verwendet, z.B. könnte als Primärschlüssel ein Attribut „KundeNr“ in der Dimensionstabelle „Kunde“ verwendet werden. Der Vorteil solcher „Surrogatschlüssel“ liegt darin, dass sie meist kürzer sind, wodurch die entsprechenden Fremdschlüssel der Fakttabelle weniger Speicherplatz benötigen. Darüber hinaus werden Abfragen einfacher, da maximal ein Attribut der Fakt- und Dimensionstabellen verglichen werden muss. [PS04] Eine Abfrage, welche eine Fakttabelle mit den Dimensionstabellen verknüpft, wird auch als „Star-Join“ bezeichnet [Inm05]. Die Effizienz von „Star-Joins“ kann durch Verwendung von Surrogatschlüsseln verbessert werden [PS04].

Betrachtet man die Abbildung 2.5, erkennt man, dass die Dimensionstabellen vom Prinzip der relationalen Normalisierung abweichen, da beispielsweise in der Dimensionstabelle „Kunde“ das Attribut „Tarif“ die Attribute „Bezeichnung“ und „Grundgebühr“ bestimmt. Negative Folgen dieser Abweichung vom Prinzip der Normalisierung sind redundante Datenspeicherung, erschwerte Datenänderungen und drohende Dateninkonsistenzen. In Data Warehouses können diese Auswirkungen oft ignoriert werden, da die Dimensionstabellen im Gegensatz zu Fakttabellen nicht speicherplatzkritisch sind und die Daten in den Dimensionstabellen meist nur selten geändert werden. [PS04]

2.4.2.2 Snowflake-Schema

Das Snowflake-Schema unterscheidet sich vom Star-Schema dadurch, dass für jede Klassifikationsstufe eine eigene Tabelle erstellt wird. Da zwischen zwei benachbarten Klassifikationsstufen eine n:1 Beziehung besteht, enthält jede solche Tabelle einen Fremdschlüssel der direkt benachbarten Klassifikationsstufe höherer Granularität. [LL03] Abbildung 2.6 zeigt ein Snowflake-Schema für die Dimension „Kunde“:

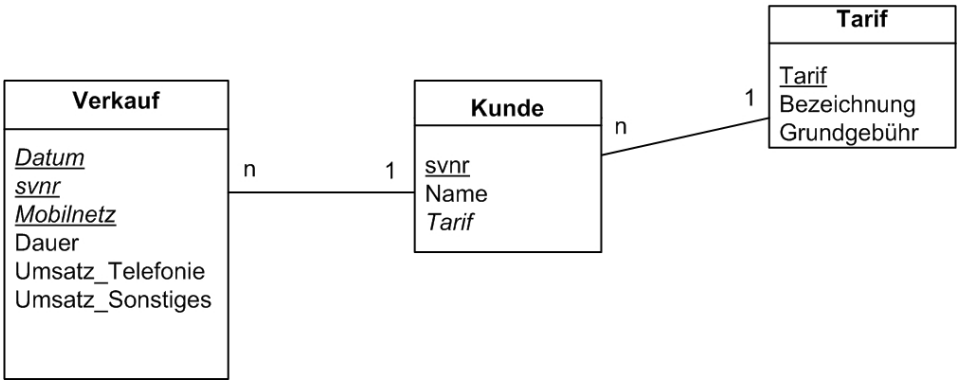


Abbildung 2.6: Snowflake-Schema: Verkauf (MFUA)

Bezüglich der Fakttabelle gibt es keinen Unterschied zwischen Star- und Snowflake-Schema, jedoch bestehen Unterschiede in der Modellierung der Dimensionen. Die Dimension „Kunde“ besteht nun aus zwei Tabellen, jeweils eine für die Klassifikationsstufe „svnr“ und „Tarif“,

wobei die Fakttabelle einen Fremdschlüssel der Klassifikationsstufe mit der niedrigsten Granularität besitzt, in der Abbildung 2.6 „svnr“. Die zwischen benachbarten Klassifikationsstufen bestehenden n:1 Beziehungen werden mittels Fremdschlüsseln dargestellt. Somit enthält die Tabelle „Kunde“ einen Fremdschlüssel der Tabelle „Tarif“. [BG04]

Im Gegensatz zum Star-Schema, werden in einem Snowflake-Schema die Dimensionstabellen normalisiert gespeichert, wodurch die für Star-Schemata charakteristischen Probleme nicht auftreten. Der größte Nachteil, welcher aus der Normalisierung der Dimensionstabellen resultiert, besteht in komplexeren „Star-Joins“, da im Vergleich zu Star-Schemata mehr Tabellen verknüpft werden müssen. [PS04]

Die Auswahl der konkreten Repräsentationsform für die Dimensionen hängt von folgenden Aspekten ab: Werden die Dimensionsinstanzen selten geändert, so ist aufgrund der effizienteren Abfrageverarbeitung das Star-Schema zu bevorzugen; sind häufige Änderungen zu erwarten, ist das Snowflake-Schema vorteilhafter. Die Verwendung des normalisierten Snowflake-Schemas ist auch dann zu erwägen, wenn verschiedene Fakttabellen unterschiedliche Klassifikationsstufen einer Dimension referenzieren (z.B. die Fakttabelle „Verkauf“ referenziert die Klassifikationsstufe „Tag“, eine andere Fakttabelle „Verträge“ referenziert die Klassifikationsstufe „Monat“). Der Grund dafür liegt darin, dass die Daten der Fakttabelle direkt auf die Relation der entsprechenden Klassifikationsstufe verweisen können und keine „on-the-fly“-Berechnungen für Aggregationen erforderlich sind. Darüber hinaus ist es möglich die beiden Ansätze zu kombinieren, d.h. einige Dimensionen werden gemäß Star-Schema modelliert, andere gemäß Snowflake-Schema. Diese Kombination wird auch als Starflake-Schema bezeichnet. [PS04]

2.4.2.3 OLAP-Abfragen und materialisierte Sichten

Nachdem das logische Schema für ein Data Warehouse festgelegt wurde, können multidimensionale OLAP-Abfragen formuliert werden. Zur Formulierung solcher Abfragen wird für ROLAP-Systeme meist SQL, für MOLAP-Systeme u.a. die von Microsoft entwickelte multidimensionale Abfragesprache MDX (Multidimensional Expressions) [MDX], insbesondere in Microsoft SQL Server 2005 Analysis Services, verwendet. Da der ursprüngliche SQL-Standard nur die Aggregatsfunktionen SUM, AVG, MIN, MAX und COUNT unterstützt, wurden die Standards SQL99 [SQLb] und SQL2003 [SQLa] um zusätzliche OLAP-Funktionalitäten, wie multiple Gruppierungskombinationen mit dem ROLLUP- oder CUBE-Operator, welcher erstmals in [GCB+97] vorgeschlagen wurde, erweitert. Eine SQL-Abfrage mit einem ROLLUP-Operator, abgesetzt auf das Star-Schema der Abbildung 2.5, könnte etwa folgendermaßen aussehen:

Beispiel 2.9: Die Abfrage des Listing 2.1 liefert als Ergebnis die durchschnittliche Gesprächsdauer pro Jahr und Tarif, die durchschnittliche Gesprächsdauer pro Jahr (über alle Tarife) sowie die durchschnittliche Gesprächsdauer über alle Dimensionen (= Ergebnis, wenn kein GROUP BY verwendet werden würde).

```
1 SELECT d.Jahr, k.Tarif, AVG(v.Gesprächsdauer)
2 FROM Verkauf v, Datum d, Kunde k
3 WHERE v.Datum = d.Datum AND v.svnr = k.svnr
4 GROUP BY ROLLUP(d.Jahr, k.Tarif)
```

Listing 2.1: SQL: OLAP-Abfrage

Der ROLLUP-Operator ermöglicht die Berechnung mehrerer Gruppierungskombinationen in einer fixierten Folge.

Da die Abfrageverarbeitung in einem Data Warehouse aufgrund der großen Datenmengen oft sehr lange Antwortzeiten mit sich bringt, werden häufige (Teil-)Abfragen mit Hilfe so genannter „materialisierter Sichten“ im Voraus berechnet und deren Ergebnisse gespeichert. Diese materialisierten Sichten können von Abfragen, deren (Teil-)Ergebnis aus diesen berechnet werden kann, verwendet werden um die Antwortzeiten zu verringern. Werden materialisierte Sichten im Data Warehouse verwendet, so müssen folgende Problembereiche beachtet werden: [BG04]

- **Verwendung materialisierter Sichten:** Werden materialisierte Sichten zur Verbesserung der Antwortzeiten verwendet, so muss deren transparente Nutzung gewährleistet sein. Das heißt, dass bestehende Sichten keinen Einfluss auf die Formulierung von Abfragen haben dürfen.
- **Auswahl materialisierter Sichten:** Da materialisierte Sichten redundante Datenspeicherung bewirken, ist eine Abstimmung zwischen der gewünschten Abfragelaufzeit und dem zusätzlich benötigten Speicherplatz vorzunehmen. Es bestehen grundsätzlich zwei Varianten zur Auswahl materialisierter Sichten, nämlich die statische Auswahl und die dynamische Auswahl. Bei der statischen Auswahl bleiben die erstellten Sichten einen bestimmten Zeitraum erhalten; bei der dynamischen Auswahl werden aktuelle Abfrageergebnisse materialisiert und in einem Cache gespeichert bis sie durch neue Ergebnisse ersetzt werden.
- **Wartung materialisierter Sichten:** Änderungen am Datenbestand erfordern eine Aktualisierung der betroffenen materialisierten Sichten.

2.4.3 Physisches Design

Im Abschnitt 2.4 wurde einleitend erwähnt, dass das logische Modell möglichst einfach bzw. automatisch in ein physisches Modell transformierbar sein muss. Dies kann für ein ROLAP-System mit der Datenbankdefinitionssprache SQL-DDL [SQLb] erfolgen, womit die modellierten Relationen und deren Beziehungen zueinander in der Datenbank angelegt werden können. Darüber hinaus sind die optimale Auswahl von Indexstrukturen sowie die Partitionierung von Relationen wichtige Aufgaben des physischen Designs.

2.4.3.1 Indexstrukturen

Bei OLAP-Abfragen wird in der Regel aus einer großen Detaildatenmenge ein bestimmter Datenbereich ausgewählt, welcher über Dimensionen eingeschränkt wird. Diese multidimensionalen (Bereichs-)Abfragen können auch als eine Menge von Restriktionen auf Klassifikationsstufen aufgefasst werden, z.B. „Monat“ in der Dimension „Datum“. [BG04]

Daher ist die optimale Auswahl von Indexstrukturen, unter Berücksichtigung des logischen Schemas und der voraussichtlichen Arbeitslast, zur Minimierung der Antwortzeiten für OLAP-Abfragen, eine wichtige Aufgabe des physischen Designs. [GR98] Gebräuchliche Indexstrukturen sind eindimensionale Baumindexstrukturen, wie z.B. „B-Baum“ und „B*-Baum“, mehrdimensionale Baumindexstrukturen, wie z.B. „R-Baum“, oder „Bitmap-Indizes“. Für genauere Informationen über die verschiedenen Indexstrukturen sowie deren Einsetzbarkeit wird auf [BG04] verwiesen.

Bei der Auswahl der Indexstrukturen wird versucht die optimale Teilmenge von Indizes bei gegebender Arbeitslast zu ermitteln. Dafür wird für jeden Indextyp eine Kostenfunktion definiert, mit welcher die Kosten der Abfrageverarbeitung berechnet werden können. Die optimale Teilmenge von Indizes ist jene, welche die Verarbeitungskosten für Abfragen minimiert. [GR98]

2.4.3.2 Partitionierung

Unter Partitionierung versteht man die Aufteilung umfangreicher Relationen, d.h. insbesondere von Fakttabellen, in kleinere Teilrelationen, die Partitionen. Die Größe und der Inhalt der Partitionen werden dabei auf die erwartete Abfrage- und Aktualisierungscharakteristik abgestimmt. Dadurch, dass die Partitionen einzeln gelesen und (durch ETL-Prozesse) geschrieben werden können, beeinflussen sich Transaktionen auf verschiedenen Partitionen nicht gegenseitig. Prinzipiell lassen sich die horizontale und vertikale Partitionierung unterscheiden. Bei der horizontalen Partitionierung wird die Tupelmenge in mehrere Teilmengen aufgeteilt, wobei jede Teilrelation die gleichen Attribute hat; bei der vertikalen Partitionierung erfolgt die Bildung von Partitionen durch Projektion. [BG04]

2.5 Zusammenfassung

In diesem Kapitel wurden die für die Thematik der Diplomarbeit relevanten theoretischen Grundlagen beschrieben. Zunächst wurden die Begriffe „Data Warehousing“, „Data Warehouse“ und „OLAP“ erläutert und abgegrenzt. Im Zuge der Beschreibung einer typischen Data Warehousing-Architektur wurden die Komponenten „Data Warehouse“, „Data Marts“ und „OLAP-Server“ eingeordnet und die Notwendigkeit einer multidimensionalen Präsentation der Analyseergebnisse verdeutlicht. Aufgrund der für Data Warehouses charakteristischen Multidimensionalität wurden die Konstrukte eines multidimensionalen Datenmodells vorgestellt.

Der Schwerpunkt dieses Kapitels lag auf der Beschreibung einer DW-Design-Methodologie, welche die Phasen konzeptuelles Design, logisches Design und physisches Design umfasst. Im Zuge des konzeptuellen Designs wird ein von Implementierungsaspekten unabhängiges multidimensionales, konzeptuelles Schema erstellt. Für diesen Zweck wurde die DFM-Notation vorgestellt. Annahmen über die Implementierungsplattform werden im logischen Design getroffen, indem ein ROLAP- oder MOLAP-Speichermodell zugrunde gelegt wird.

Im in der Praxis häufiger anzutreffenden ROLAP-Speichermodell wird ein Würfel durch eine Menge von Relationen bzw. Tabellen dargestellt. In diesem Zusammenhang wurden zwei Modellierungsansätze, nämlich Star-Schema und Snowflake-Schema, präsentiert, wobei ersterer vom Prinzip der relationalen Normalisierung abweicht. Basierend auf dem konstruierten logischen Modell, können multidimensionale OLAP-Abfragen für Analysezwecke formuliert werden. Die Antwortzeiten für Abfragen können mit Hilfe von materialisierten Sichten verbessert werden.

Das physische Design beschäftigt sich mit der Umsetzung des logischen Modells auf eine Implementierungsplattform, mit der Wahl geeigneter Indexstrukturen und der Partitionierung von Relationen um die Antwortzeiten für OLAP-Abfragen zu minimieren.

Kapitel 3

Integration von Datenbank- und Data Warehouse-Systemen

Nachdem im vorherigen Kapitel die theoretischen Grundlagen des Data Warehousing erläutert wurden, ist es nun notwendig die Perspektive auf verteilte Systeme zu erweitern, um die im Kapitel 1 angesprochene Problematik der Integration von Data Warehouse-Systemen zu betrachten. Dafür ist es jedoch zunächst erforderlich die Begriffe „integriertes System“ und „Komponentensystem“ zu definieren, welche in diesem Kapitel und in den folgenden Kapiteln verwendet werden:

Definition 3.1. Ein *integriertes System* ist als logische Schicht einer Software-Architektur zu verstehen, welche Transparenz bezüglich der zwischen den Komponentensystemen (vgl. Definition 3.2) bestehenden Heterogenitäten schafft. Je nachdem, ob die Komponentensysteme Datenbanksysteme (DBS) oder Data Warehouse-Systeme (DWS) darstellen, spricht man von einem *integriertem DBS* bzw. einem *integriertem DWS*.

Mit den Begriffen der Definition 3.1 sind keine Informationen über die Art der Integration assoziiert, z.B. ob die Integration mit einem globalen Schema erfolgt.

Definition 3.2. Als *Komponentensystem* wird ein System bezeichnet, welches Bestandteil eines integrierten Systems (vgl. Definition 3.1) ist, d.h. welches mit einer übergeordneten, logischen Schicht integriert wurde. Je nach dem, ob es sich dabei um ein DBS oder DWS handelt, spricht man von einem *Komponenten-DBS* bzw. einem *Komponenten-DWS*.

Auf dem Gebiet der Integration von DBS wurde bereits viel Forschungsarbeit geleistet (vgl. [SL90] für einen Überblick). Im Gegensatz dazu wurde die Problematik der Integration von DWS von der Forschung noch nicht in dem Ausmaß erkannt. Dieses Kapitel gibt einen Überblick über den Stand der Forschung auf dem Gebiet der Integration von DBS und DWS. Um die logische Gliederung dieses Kapitels zu verstehen sind zwei weitere Definitionen erforderlich:

Definition 3.3. *Datenintegration* beschreibt das Problem, „Daten verschiedener Quellen zu kombinieren um dem Benutzer eine einheitliche Sicht dieser Daten zu präsentieren“. [Len02]

Der Begriff „Datenintegration“ ist allgemeiner Natur, d.h. unter ihm sind Ansätze zu subsummieren, welche unabhängig von den konkreten, zu integrierenden, datenhaltenden Komponentensystemen bzw. unabhängig vom zu schaffenden integrierten System sind. Darüber hinaus lässt sich aus der Definition 3.3 ableiten, dass das integrierte System auf einem globalen Schema und einer Menge von Datenquellen basiert. [Len02] Es werden jedoch keine Annahmen über die Art der Datenquellen und die Art des integrierten Systems getroffen.

Definition 3.4. *Integration von DBS/Integration von DWS* (in Anlehnung an [ÖV99]) ist definiert als die „Kombination von Information aus den teilnehmenden DBS/DWS zu einer konzeptuellen Definition eines Multi-DBS/Multi-DWS“.

Unter den Begriffen „Integration von DBS“ und „Integration von DWS“ der Definition 3.4 sind Ansätze zu subsummieren, welche Annahmen über die zu integrierenden, datenhaltenden Systeme (DBS oder DWS) bzw. über das zu schaffende integrierte System (integriertes DBS oder integriertes DWS) treffen. Darüber hinaus lässt sich ableiten, dass die Integration mit einem konzeptuellen, globalen Schema erfolgt.

Dieses Kapitel behandelt zum einen allgemeine Ansätze der Datenintegration und des weiteren Ansätze für die Integration von DBS bzw. für die Integration von DWS. Im folgenden Abschnitt werden zwei allgemeine Ansätze zur Datenintegration, nämlich „Local as view“ und „Global as view“, diskutiert. Abschnitt 3.2 behandelt die Integration von DBS, da viele Konzepte der DBS-Integration auch auf DWS übertragbar sind. Abschnitt 3.3 betrachtet die Integration von DWS, insbesondere die Problematik der Heterogenität. Der letzte Abschnitt 3.4 stellt eine Zusammenfassung des Kapitels dar.

3.1 Allgemeine Ansätze für die Datenintegration

Die in diesem Abschnitt vorgestellten zwei Ansätze zur Datenintegration (vgl. [Len02]), nämlich „Global as view“ und „Local as view“, beschreiben die Schaffung eines „Daten-Integrationssystems“ mit einem globalen, integrierten Schema. Die Datenintegration basiert bei beiden Ansätzen auf der Definition so genannter Mappings.

Definition 3.5. *Mappings* sind Funktionen, welche Objekte eines Schemas zu Objekten eines anderen Schemas in Beziehung setzen. [SL90]

Die zwei Ansätze stellen unterschiedliche Strategien für die Spezifikation von Mappings zwischen globalem Schema und den Quellschemata dar. Beide Ansätze erfordern einen Reformulierungsschritt bei Abfragen, d.h. eine Abfrage auf das globale Schema muss in eine Menge von Abfragen auf die Quellschemata transformiert werden. Im Folgenden werden die Charakteristika der beiden Ansätze sowie die Unterschiede bei der Verarbeitung von Abfragen erläutert. Zuvor ist es jedoch erforderlich das verwendete Bezugssystem zu formalisieren: [Len02]

Definition 3.6. Ein *Daten-Integrationssystem* I ist ein Tripel (G, S, M), wo

- G das globale Schema darstellt, welches in einer Sprache L_G über dem Alphabet A_G definiert ist, wobei A_G ein Symbol für jedes Element in G enthält.
- S das Quellschema darstellt, welches in einer Sprache L_S über dem Alphabet A_S definiert ist, wobei A_S ein Symbol für jedes Element in S enthält.
- M das Mapping zwischen G und S darstellt, bestehend aus einer Menge von Aussagen der Form $q_S \mapsto q_G$, $q_G \mapsto q_S$, wobei
  - q_S eine Abfrage bzw. Sicht auf das Quellschema S repräsentiert. Eine Abfrage bzw. Sicht q_S wird in einer Abfragesprache $L_{M,S}$ über dem Alphabet A_S definiert.
  - q_G eine Abfrage bzw. Sicht auf das globale Schema G repräsentiert. Eine Abfrage bzw. Sicht q_G wird in einer Abfragesprache $L_{M,G}$ über dem Alphabet A_G definiert.
  - Eine Aussage $q_S \mapsto q_G$ spezifiziert, dass das von der Abfrage q_S repräsentierte Konzept jenem der Abfrage q_G gleicht (umgekehrt für eine Aussage $q_G \mapsto q_S$).

Ein Quellschema beschreibt die Struktur einer Datenquelle, das globale Schema eine abgestimmte, integrierte und virtuelle Sicht auf die darunter liegenden Datenquellen. Die Mappings bilden eine Verbindung zwischen Objekten des globalen Schemas und den Objekten der Quellschemata. [Len02]

3.1.1 Local as view (LAV)

Ein Daten-Integrationssystem $I = (G, S, M)$, welches auf dem LAV-Ansatz basiert, ist dadurch charakterisiert, dass jedem Element s des Quellschemas S eine Sicht auf das globale Schema q_G zugeordnet wird, d.h. es wird ein Mapping der Form $s \mapsto q_G$ definiert. Dieser Ansatz ist insbesondere dann empfehlenswert, wenn das integrierte System, d.h. das globale Schema, stabil und in der Organisation etabliert ist. Der wesentliche Vorteil des LAV-Ansatzes ist die leichte Erweiterbarkeit, da eine neue Datenquelle einfach durch Definition neuer Mappings integriert werden kann.

Als Nachteil dieses Ansatzes ist anzuführen, dass sich die Verarbeitung von Abfragen in einem LAV-System sehr komplex gestalten kann, da sich die Information über die Daten des globalen Schemas auf die den Datenquellen zugeordneten Sichten beschränkt („Problematik der lückenhaften Information“ [vdM98]). Daher ist es schwierig festzustellen, wie die Datenquellen benutzt werden müssen um eine auf das globale Schema formulierte Abfrage zu beantworten. Das Problem der Abfrage-Verarbeitung in einem LAV-System wird auch als „Sichten-basierte Abfrage-Verarbeitung“ bezeichnet, da die Antwortberechnung auf einer Menge von Sichten basiert und nicht auf den Rohdaten der einzelnen Datenbanken [LMSS95] [Ull97]. [Len02]

3.1.2 Global as view (GAV)

Ein Daten-Integrationssystem $I = (G, S, M)$, welches auf dem GAV-Ansatz basiert, ist dadurch charakterisiert, dass jedem Element g des globalen Schemas G eine Sicht auf die Quellschemata q_S zugeordnet wird, d.h. es wird ein Mapping der Form $g \mapsto q_S$ definiert. Diese Art des Mappings beschreibt, wie die Daten für das globale Schema abgerufen werden können bzw. müssen und ist dann empfehlenswert, wenn die Quellschemata stabil sind. Im Gegensatz zum LAV-Ansatz gestaltet sich jedoch die Erweiterung um neue Datenquellen als schwierig, da eine neue Quelle die Mappings von Elementen des globalen Schemas beeinflussen kann, wodurch die Sichten ggf. neu definiert werden müssen.

Der Vorteil dieses Ansatzes liegt in der grundsätzlich einfachen Abfrage-Verarbeitung, da mit den Mappings festgelegt ist, wie die Daten für ein Element des globalen Schemas aus den Quellschemata abgerufen werden müssen. „Grundsätzlich“ deshalb, weil die Annahme zugrunde liegt, dass alle Sichten exakt sind, d.h. $q_S = g$, und keine Zusicherungen im globalen Schema definiert wurden. Sind diese Annahmen nicht gültig, gestaltet sich die Verarbeitung von Abfragen auch komplexer. [Len02]

3.1.3 Deklarative vs. prozedurale Integration

Die Art der Beschreibung der Mappings, deklarativ oder prozedural, steht in engem Zusammenhang mit den zuvor erläuterten Ansätzen. Bei der deklarativen Integration werden die Mappings in Form von Regeln definiert, es geht also um das „Was?“. Diese Art der Integration ist im LAV-Ansatz aufgrund der Frage-Antwort-Kombination „Was sind die Elemente der Quellschemata? Sichten auf die Elemente des globalen Schemas!“ vorherrschend. Bei der prozeduralen Integration dominiert der mit der Integration verbundene Ablauf, d.h. das „Wie?“. Diese Art der Integration ist im GAV-Ansatz ersichtlich, da die Frage „Wie werden die Daten aus den Quellschemata abgerufen?“ beantwortet wird. [DG06]

3.2 Ansätze für die Integration von Datenbanksystemen

Gemäß Definition 3.4 geht es bei der Integration von DBS um die Schaffung eines aus mehreren DBS bestehenden, integrierten DBS, wobei die Integration mit einem globalen Schema erfolgt. In diesem Abschnitt werden zunächst verschiedene Typen von integrierten DBS, nämlich „Verteilte DBS“, „Nicht-Föderierte DBS“, „Föderierte DBS“ und „Multi-DBS“, mit ihren spezifischen Charakteristika klassifiziert. In diesem Zusammenhang ist anzumerken, dass nicht alle Typen von integrierten DBS der Definition 3.4 entsprechen, da nicht alle ein globales Schema verwenden. Integrierte DBS mit globalem Schema können auf einem der beiden vorgestellten, allgemeinen Ansätze der Datenintegration, LAV oder GAV (vgl. Abschnitt 3.1),

basieren. Im Anschluss wird ein Architekturmodell eines integrierten DBS präsentiert, bevor die Problematik der Heterogentität bei der Integration von DBS diskutiert wird.

3.2.1 Klassifizierung von Systemen zur Datenbankintegration

Systeme, welche aus mehreren Datenbanken bestehen, können anhand der Dimensionen „Verteilung“, „Heterogenität“ und „Autonomie“ charakterisiert werden [SL90]:

- „Verteilung“ beschreibt die Eigenschaft, dass Datenbanken auf mehreren, möglicherweise geographisch verteilten, über ein Netzwerk verbundenen Rechnern installiert sind.
- „Heterogenität“ beschreibt Unterschiede bezüglich Hardware, Betriebssystem, etc., Unterschiede im verwendeten Datenbankmanagementsystem (DBMS) sowie semantische Unterschiede auf Schema- und Instanzebene. Die zuletzt angeführten semantischen Unterschiede sind von besonderer Bedeutung und werden im Abschnitt 3.2.3 genauer behandelt.
- „Autonomie“ beschreibt die Eigenschaft, dass eine Datenbank unabhängig von anderen Datenbanken des verteilten Systems entwickelt und betrieben wird.

Ein „Zentrales DBS“ besteht aus einem DBMS, welches eine einzige Datenbank auf demselben Rechner verwaltet [SL90]. Die für ein Zentrales DBS charakteristischen Dimensionsausprägungen sind nicht-verteilt, homogen und autonom.

Als „Verteilte DBS“ werden jene Systeme bezeichnet, wo ein verteiltes DBMS (DDBMS) mehrere, möglicherweise auf unterschiedlichen Rechnern installierte Datenbanken verwaltet und Verteilungstransparenz für die Benutzer gewährleistet [ÖV99]. Die für ein Verteiltes DBS charakteristischen Dimensionsausprägungen sind verteilt, homogen/heterogen und nicht-autonom.

Im Gegensatz dazu bezeichnet man einen Verbund von mehreren DBMS als „Multidatenbanksysteme“ [Con97], welche wiederum anhand unterschiedlicher Dimensionsausprägungen präziser klassifiziert werden können. Da sich in der Literatur keine der vorgeschlagenen Klassifizierungen von Multidatenbanksystemen wirklich durchgesetzt hat, wird im Folgenden jene in [SL90] vorgeschlagene erläutert.

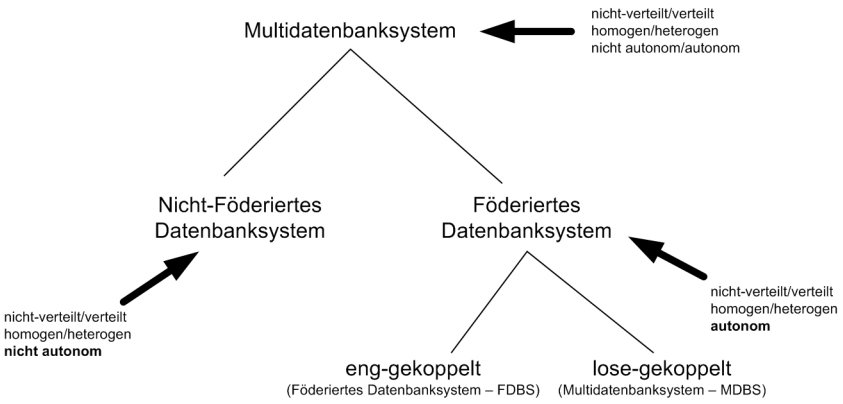


Abbildung 3.1: Klassifizierung: Multidatenbanksysteme (vgl. [SL90])

Die Wurzel der Abbildung 3.1 bildet „Multidatenbanksystem“, welches mehrere Komponenten-DBS umfasst. Jedes Komponenten-DBS wird von einem Komponenten-DBMS verwaltet. Die Komponenten-DBS können wiederum als Zentrale oder Verteilte DBS realisiert sein. Multidatenbanksysteme können nicht-verteilt oder verteilt, homogen oder heterogen und autonom oder nicht-autonom sein. Anhand der Dimension „Autonomie“ lassen sich Multidatenbanksysteme präziser klassifizieren.

Ein „Nicht-Föderiertes DBS“ stellt eine Unterkategorie von Multidatenbanksystemen dar und integriert nicht-autonome Komponenten-DBS. Ein derartiges System wird zentral verwaltet und alle Operationen werden einheitlich ausgeführt. Werden alle Komponenten-DBS vollständig in ein globales Schema integriert, ähnelt dieses System logisch einem Verteilten DBS.

Im Gegensatz dazu besteht ein „Föderiertes DBS“ aus autonomen Komponenten-DBS, welche ihre Daten in der Föderation teilweise zugänglich machen ohne bestehende Anwendungen zu beeinflussen. Diese Föderierten DBS können zusätzlich hinsichtlich ihres Kopplungsgrads wie folgt präziser klassifiziert werden:

Ein Föderiertes DBS wird als „lose-gekoppelt“ bezeichnet, wenn kein globales, integriertes Schema existiert und somit die Integration der Komponenten-DBS dem Aufgabenbereich der Benutzer zuzuordnen ist. Jeder Benutzer kann somit ein eigenes „föderiertes Schema“ durch Definition entsprechender Sichten auf die Komponenten-DBS, ggf. durch Verwendung einer „Multidatenbank-Abfragesprache“, erstellen. Ein Überblick über derartige Abfragesprachen folgt im Abschnitt 4.2. Ein lose-gekoppeltes Föderiertes DBS wird oft auch als Multidatenbanksystem (MDBS) bezeichnet [LMR90]. Wenn im Folgenden von Multidatenbanksystemen bzw. MDBS die Rede ist, ist ein lose-gekoppeltes Föderiertes DBS gemeint und nicht die gleichlautende, übergeordnete Kategorie (vgl. Abbildung 3.1). Abschließend sei noch kritisch angemerkt, dass in einem MDBS hohe Anforderungen an die Benutzer gestellt werden, da für die Integrationsaufgaben umfangreiche Kenntnisse über Datenbanken und deren Integration, insbesondere über die Handhabung von Heterogenitäten, erforderlich sind.

Ein „eng-gekoppeltes“ Föderiertes DBS ist durch das Vorhandensein eines oder mehrerer globaler, von einem Administrator der Föderation entwickelter Schemata gekennzeichnet, welche die Komponenten-DBS integrieren und Heterogenitäten auflösen. Der Benutzer kann in einem solchen System dieselben Abfragesprachen verwenden wie in Zentralen oder Verteilten DBS, da Orts-, Verteilungs- und Replikationstransparenz gegeben sind [SL90]. In den folgenden Ausführungen wird diese enger gefasste Definition mit einem Föderierten DBS (FDBS) assoziiert (vgl. Abbildung 3.1).

3.2.2 Architektur eines Föderierten Datenbanksystems

Abbildung 3.2 visualisiert die in [SL90] vorgeschlagene Fünf-Ebenen-Schema-Architektur für ein FDBS:

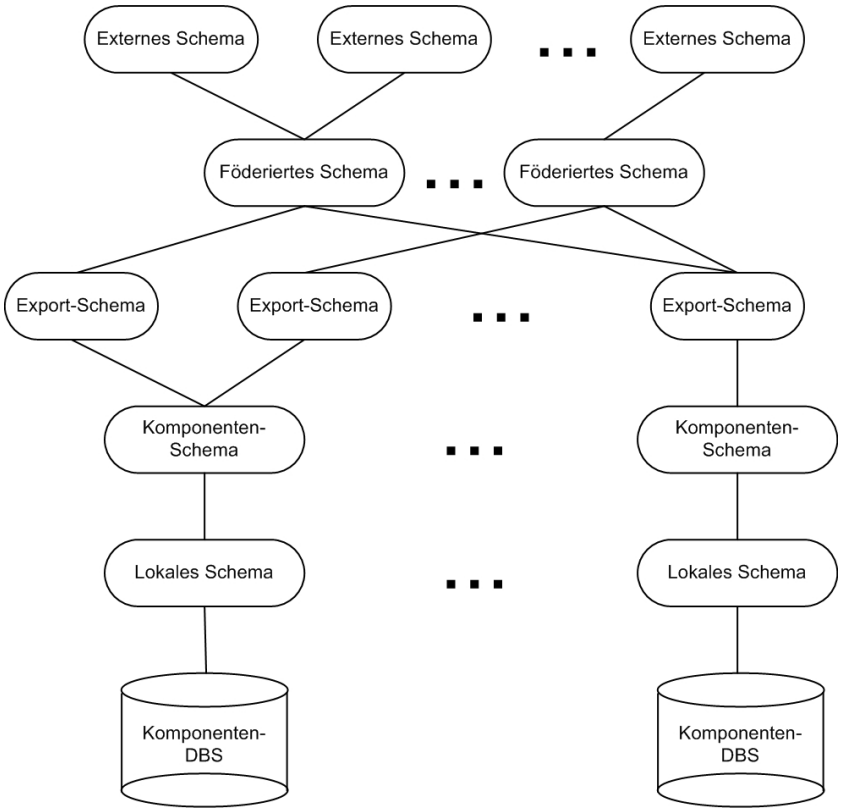


Abbildung 3.2: Fünf-Ebenen-Schema-Architektur für FDBS (vgl. [SL90])

Lokales Schema:
Das lokale Schema ist das konzeptuelle Schema eines Komponenten-DBS. Es ist im proprietären Datenmodell des jeweiligen DBMS definiert.

Kompontenten-Schema:
Ein Komponenten-Schema wird aus dem lokalen Schema abgeleitet, indem das proprietäre Datenmodell in das kanonische Datenmodell des FDBS übersetzt wird.

Export-Schema:
Da nicht alle Daten der Komponenten-DBS den Benutzern der Föderation zugänglich gemacht werden sollen, repräsentiert das Export-Schema jene Teile der Komponenten-Schemata, welche im FDBS verfügbar sein sollen.

Föderiertes Schema:
Das föderierte Schema integriert ein oder mehrere Export-Schemata und enthält zusätzlich Information über die Datenverteilung, wodurch Verteilungstransparenz gewährleistet wird.

Externes Schema:
Ein externes Schema definiert ein Schema für Benutzer und/oder Anwendungen, wodurch z.B. Zugriffskontrollen realisiert werden können.

Das Architekturmodell der Abbildung 3.2 gewährleistet die für FDBS charakteristische Autonomie, da bestehende Anwendungen weiter auf den lokalen Schemata der Komponenten-DBS aufsetzen können. Anwendungen oder Benutzer, welche integrierte Ergebnisse aus mehreren Komponenten-DBS erfordern, definieren Abfragen auf das FDBS bzw. auf ein oder mehrere föderierte oder externe Schemata. Das föderierte Schema schirmt dabei die Anwendungen oder Benutzer von der Datenverteilung (Verteilungstransparenz) sowie von bestehenden Heterogenitäten ab. [SL90]

Heterogenitäten zwischen den einzelnen Komponenten-DBS erschweren die Integration, d.h. die Erstellung eines globalen Schemas (für FDBS) bzw. die Erstellung von Sichten und Abfragen durch die Benutzer (für MDBS). Der nächste Abschnitt gibt einen Überblick über mögliche, bei der Integration von DBS auftretende Heterogenitäten.

3.2.3 Heterogenitäten zwischen Datenbanksystemen

Im Zuge der Integration von DBS müssen Heterogenitäten auf verschiedenen Ebenen berücksichtigt werden: [SL90]

- Hardware (Befehlssätze, Datenformate, Konfigurationen, etc.)
- Betriebssystem (Dateisysteme, Dateitypen, Inter-Prozesskommunikation, etc.)
- Datenbanksystem
  - DBMS (Datenmodelle, Abfragesprachen, Zugriffskontrolle, etc.)
  - semantische Heterogenität

Für die Schemaintegration (für FDBS) bzw. die Erstellung von Sichten und Abfragen durch die Benutzer (für MDBS) sind vor allem semantische Heterogenitäten von Bedeutung, weshalb deren Problematik nun näher erläutert wird.

Semantische Heterogenität
Semantische Heterogenität meint Variationen in der Art, wie Daten in verschiedenen Datenbanken abgebildet sind. Sie ist eine natürliche Konsequenz aus der Tatsache, dass Datenbanken oft voneinander unabhängig (weiter-)entwickelt werden um die spezifischen Anforderungen der auf ihnen aufsetzenden Anwendungen zu erfüllen [HM93]. Aufgrund semantischer Heterogenitäten auf Schemaebene (betreffend Metadaten, insbesondere Schemaattribute) und Instanzebene (betreffend Daten) können folgende Konflikte bei der Integration von DBS auftreten: [Bre90]

- **Namenskonflikte**, wenn semantisch identische Daten bzw. Metadaten unterschiedlich benannt (Synonyme) bzw. wenn semantisch unterschiedliche Daten bzw. Metadaten gleich benannt werden (Homonyme).
- **Konflikte aufgrund unterschiedlicher Datenrepräsentationen**, wenn für semantisch identische Daten unterschiedliche Datentypen verwendet werden.
- **Domänenkonflikte**, wenn semantisch identische Daten in unterschiedlichen Einheiten gespeichert werden (z.B. unterschiedliche Währungen bei Beträgen).
- **Konflikte aufgrund fehlender oder widerspüchlicher Daten**, wenn semantisch identische Datenobjekte in manchen Datenquellen unterschiedliche oder fehlende Werte aufweisen.
- **Schema-Instanz-Konflikte**, wenn semantisch identische Datenobjekte in einer Datenquelle als Daten und in anderen Datenquellen als Schemaattribute bzw. Metadaten modelliert sind [LSS01].

3.2.4 Lösungsansätze

Die Möglichkeiten, wie die eben angeführten Konflikte im Zuge der Integration von DBS aufgelöst werden können, hängen davon ab, welche Art eines integrierten DBS man wählt. FDBS werden unter anderem in [SL90], [LMR90] und [Con97] behandelt:

In [SL90] werden unterschiedliche Architekturmodelle für FDBS vorgestellt, welche auf der im Abschnitt 3.2.2 behandelten Fünf-Ebenen-Schema-Architektur basieren. Der Entwurf eines FDBS wird in [Con97] erläutert, wobei der Schwerpunkt auf der Schemaintegration liegt. In diesem Zusammenhang wird gezeigt wie semantische Heterogenitäten gehandhabt werden können um ein globales, integriertes Schema zu erstellen. Die Problematik der verteilten Abfrageverarbeitung in einem FDBS wird in [LMR90] und [Con97] erörtert, wobei in [Con97] vertieft auf die Transaktionsverwaltung in einem FDBS eingegangen wird. In diesem Kontext werden die Probleme der Transaktionsverwaltung in einem FDBS, welche aus der Autonomie der Komponentensysteme resultieren, skizziert und Lösungsvorschläge präsentiert. Darüber hinaus wurde für FDBS bereits eine Reihe von Ansätzen und Werkzeugen entwickelt, welche

die (semi-)automatische Erzeugung eines globalen Schemas unterstützen (vgl. [RB01] für einen Überblick).

Für MDBS wurden vor allem spezielle Abfragesprachen entwickelt, welche für die Definition von integrierten Sichten auf die Komponenten-DBS herangezogen werden können. Der Abschnitt 4.2 gibt einen Überblick über die wichtigsten entwickelten Abfragesprachen, wobei der Fokus der Beschreibungen auf einem möglichen Einsatz für die Integration von DWS liegt.

3.3 Ansätze für die Integration von Data Warehouse-Systemen

Analog zum Abschnitt 3.2 wird zunächst erläutert, inwieweit sich in der Forschung eine Klassifizierung unterschiedlicher Typen von integrierten DWS durchgesetzt hat. Im Anschluss wird ein Architekturmodell eines ausgewählten integrierten DWS vorgestellt, bevor auf die Problematik der Heterogenität bei der Integration eingegangen wird.

3.3.1 Klassifizierung von Systemen zur Data Warehouse-Integration

In [ABJ+03] wird ein „Verteiltes DWS“ als eine „Sammlung von nicht-autonomen, lokalen Data Warehouses zusammen mit einem Koordinatorsystem, welches die Ergebnisse der Sub-Abfragen korelliert“, definiert. Da das Koordinatorsystem einem verteilten DBMS ähnelt, ist ersichtlich, dass diese Definition von dem eines Verteilten DBS abgeleitet wurde.

Für die Integration von unabhängig entwickelten, autonomen DWS hat sich noch keine Klassifizierung durchgesetzt, was zum Teil auch daran liegen dürfte, dass die Problematik der Integration von DWS erst heutzutage relevant wird. In [BS06a] wurde eine Klassifizierung von Systemen zur Integration von DWS, analog zu der für Datenbanken im Abschnitt 3.2.1 vorgestellten, verwendet.

Demnach ist ein „Multi-Data Warehouse-System (MDWS)“ durch eine lose Kopplung gekennzeichnet, da auf die Komponenten-DWS direkt zugegriffen wird und die Integrationsaufgabe dem Benutzer zukommt, d.h. es existiert kein globales Schema zur Integration der Komponenten-Würfel.

Ein „Föderiertes Data Warehouse-System (FDWS)“ ist eng-gekoppelt und bietet transparenten Zugriff auf die Komponenten-DWS durch die Bereitstellung eines integrierten, globalen Schemas, welches im DW-Kontext als globaler Würfel bezeichnet wird.

Im folgenden Abschnitt wird die Architektur eines eng-gekoppelten FDWS erläutert.

3.3.2 Architektur eines Föderierten Data Warehouse-Systems

Basierend auf der Fünf-Ebenen-Schema-Architektur [SL90] für FDBS wird in [BS08] ein Architekturmodell für ein eng-gekoppeltes FDWS vorgeschlagen. In diesem Modell erfolgt die Integration mit einem globalen, multidimensionalen Schema, welches von den zwischen den Komponenten-DWS bestehenden Heterogenitäten abstrahiert.

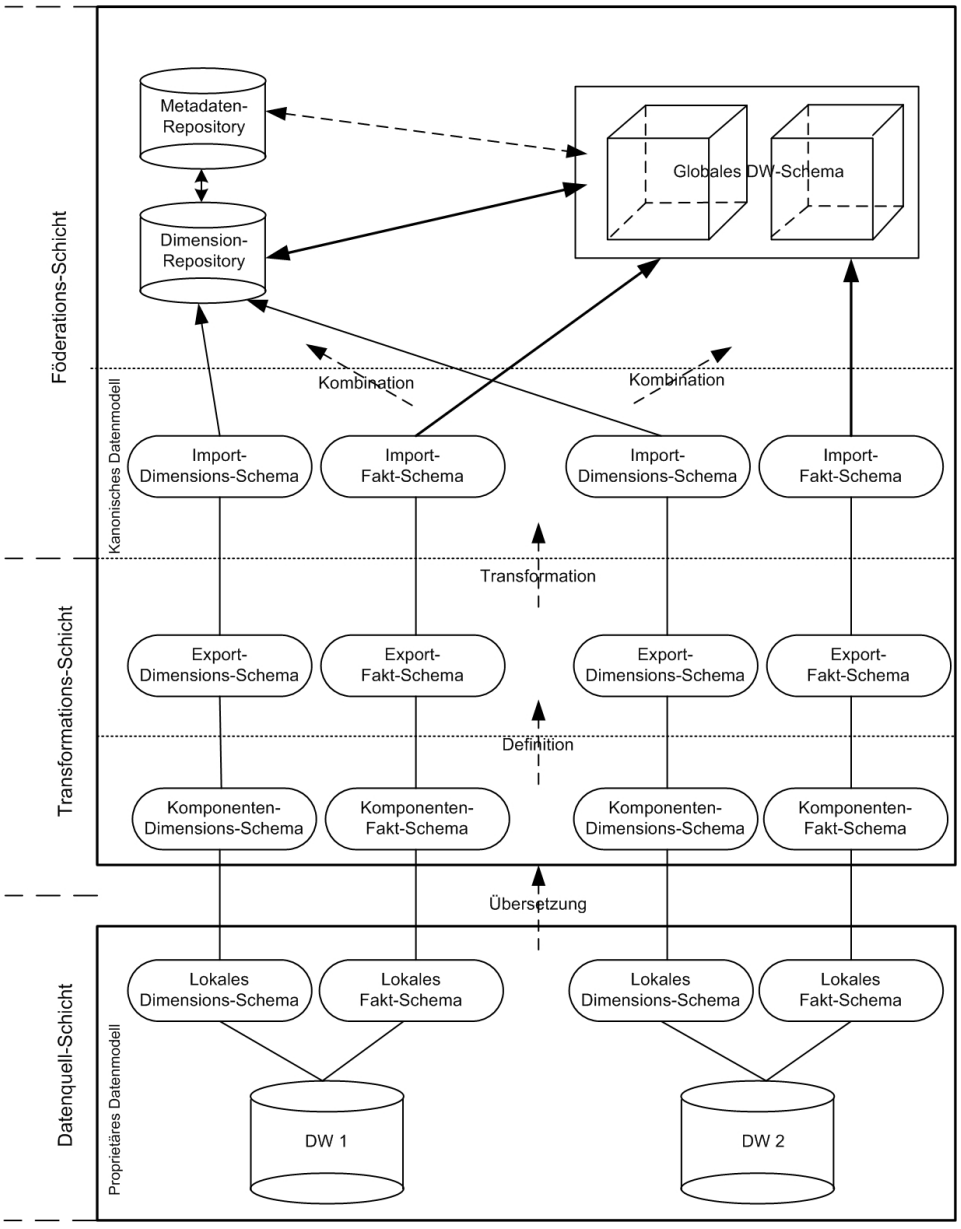


Abbildung 3.3: Architektur eines FDWS (vgl. [BS08])

Wie in Abbildung 3.3 ersichtlich, enthält das Architekturmodell eines FDWS fünf Schema-Ebenen, von welchen vier bereits in Abschnitt 3.2.2 kurz erläutert wurden. Die Import-Schemata

stellen konfliktbereinigte Versionen der Export-Schemata dar. Das globale DW-Schema bezeichnet das föderierte Schema des FDWS.

Der Prozess der Integration der Komponenten-DWS kann in vier Phasen gegliedert werden:

- In der Phase „Übersetzung“ erfolgt eine Transformation der in einem proprietären Datenmodell definierten lokalen Schemata in das kanonische Datenmodell des FDWS (=Komponenten-Schemata).
- In der Phase „Definition“ werden aus den Komponenten-Schemata Export-Schemata erzeugt, welche die im globalen Schema zu integrierenden Teile der Dimensions- und Fakt-Schemata enthalten.
- In der Phase „Transformation“ erfolgt eine Überführung der Export-Schemata in Import-Schemata. In diesem Schritt werden die zwischen den Schemata bestehenden Heterogenitäten (vgl. Abschnitt 3.3.3) aufgelöst. Um die Heterogenitäten zu beseitigen ist ein Formalismus zur Definition von Mappings zwischen den Schemata erforderlich. Aus diesem Grund werden im Abschnitt 4.3 sowie im Kapitel 5 diverse Abfragesprachen hinsichtlich ihrer Einsetzbarkeit zur Auflösung der Heterogenitäten analysiert.
- In der Phase „Kombination“ werden aus den Import-Dimensions-Schemata globale Dimensionen erzeugt und im „Dimension-Repository“ abgelegt. Die Erzeugung des globalen DW-Schemas erfolgt aus den Import-Fakt-Schemata und den globalen Dimensionen des Dimension-Repository.

In dieser Architektur eines FDWS wird die „Föderations-Schicht“ verwendet um den Benutzer von den zwischen den Schemata der Komponenten-DWS bestehenden Heterogenitäten abzuschirmen und um transparenten Zugriff auf die Komponenten-DWS zu gewährleisten. Folgedessen erscheint dem Benutzer das FDWS wie ein einzelnes Data Warehouse.

Bei der Integration von DWS können im Vergleich zur Integration von DBS zusätzliche Heterogenitäten auftreten, da das einem Data Warehouse zugrunde liegende, multidimensionale Datenmodell (vgl. Abschnitt 2.3) semantisch reichhaltiger ist als das relationale Datenmodell [Cod70]. Das relationale Datenmodell kennt im Wesentlichen nur Entitäten während im multidimensionalen Datenmodell Fakten (Fakt-Entitäten) und Dimensionen (Dimensions-Entitäten) mit Hierarchien existieren. Der folgende Abschnitt widmet sich der Erläuterung der semantischen Heterogenitäten, die bei der Integration von DWS auftreten können. Im Abschnitt 3.3.4 erfolgt eine Gegenüberstellung der semantischen Heterogenitäten zwischen DBS (vgl. Abschnitt 3.2.3) und der semantischen Heterogenitäten zwischen DWS (vgl. Abschnitt 3.3.3).

3.3.3 Heterogenitäten zwischen Data Warehouse-Systemen

Die folgende Tabelle gliedert mögliche Heterogenitäten auf Schema- und Instanzebene nach der betroffenen multidimensionalen Entität, d.h. nach Fakten und Dimensionen:

	Fakten	**Dimensionen**
Schemaebene	- Dimensionalität - Namenskonflikte (Kenngrößen) - Domänenkonflikte (Kenngrößen)	- Unterschiedliche Aggregationshierarchien - Domänenkonflikte (innere Klassifikationsstufe) - Domänenkonflikte (unterste Klassifikationsstufe) - Domänen und/oder Namenskonflikte ((nicht-)dimensionale Attribute/ Klassifikationsstufen)
Schema vs. Instanz	- Fakt-Kontext als Dimensionsinstanz	- Dimensionsinstanzen als kontextualisierte Fakten
Instanzebene	- Überlappende/Disjunkte Fakten	- Namenskonflikte - Überlappende/Disjunkte Dimensionsinstanzen - Heterogene roll-up Beziehungen

Tabelle 3.1: Semantische Heterogenitäten zwischen DWS (vgl. [BS06a])

Im Folgenden werden die in der Tabelle 3.1 angeführten Heterogenitäten auf Schemaebene, auf Schema-Instanz-Ebene sowie auf Instanzebene erörtert, bevor eine Gegenüberstellung mit den im Abschnitt 3.2.3 identifizierten Heterogenitäten zwischen DBS erfolgt.

3.3.3.1 Konflikte auf Schemaebene

Die Abbildung 3.4 zeigt die beiden konzeptuellen DW-Schemata für die Mobilfunkunternehmen MFUA und MFUB:

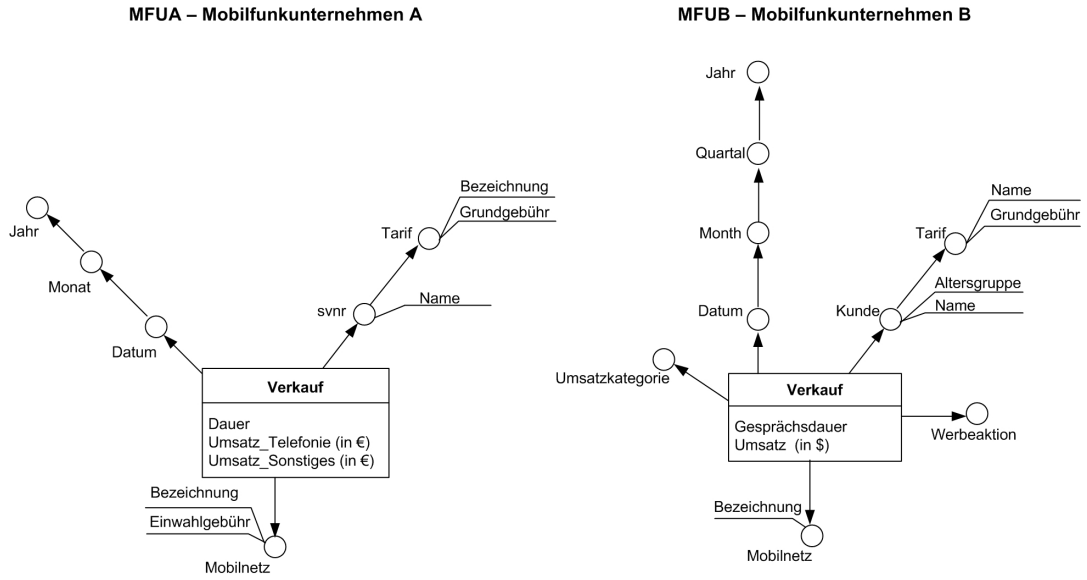


Abbildung 3.4: Rahmenbeispiel: Konzeptuelle DW-Schemata

Die beiden Würfel speichern ähnliche Daten, jedoch in heterogenen Schemata. Im Folgenden werden die möglichen Heterogenitäten auf Schemaebene bzw. auf Schema-Instanz-Ebene (vgl. Tabelle 3.1) erläutert: [BS06b]

Dimensionalität
Ein Dimensionalitäts-Konflikt ist auf eine unterschiedliche Anzahl von Dimensionen in den Faktschemata zurückzuführen.

Beispiel 3.1: Die Dimension „Werbeaktion“ des MFUB-Schemas ist im MFUA-Schema nicht modelliert.

Unterschiedliche Aggregationshierarchien
Unterschiedliche Aggregationshierarchien bestehen, wenn zwei oder mehrere semantisch äquivalente Dimensionen aufgrund einer unterschiedlichen Anzahl bzw. unterschiedlicher Domänen von Klassifikationsstufen inkompatibel sind.

Beispiel 3.2: Die Dimension „Datum“ besitzt im MFUA-Schema drei Klassifikationsstufen, nämlich „Datum“, „Monat“ und „Jahr“, im MFUB-Schema jedoch vier Klassifikationsstufen, nämlich „Datum“, „Month“, „Quartal“ und „Jahr“.

Namenskonflikte
Namenskonflikte treten auf, wenn für semantisch äquivalente Schemaattribute unterschiedliche Bezeichnungen verwendet werden. Demnach können Namenskonflikte bei Kenngrößen, Klassifikationsstufen und (nicht-)dimensionalen Attributen auftreten.

Beispiel 3.3: Im MFUA-Schema wird die Dauer von Telefongesprächen im Kenngrößen-Attribut „Dauer“ gespeichert, im MFUB-Schema im Attribut „Gesprächsdauer“. Weitere Namenskonflikte betreffen die Klassifikationsstufe „Monat“ der Dimension „Datum“ im MFUA-Schema, welche im MFUB-Schema „Month“ heißt, das dimensionale Attribut „svnr“ des MFUA-Schemas, welches im MFUB-Schema auf „Kunde“ lautet und das nicht-dimensionale Attribut „Bezeichnung“ der Klassifikationsstufe „Tarif“ des MFUA-Schemas, welches im MFUB-Schema dem nicht-dimensionalen Attribut „Name“ entspricht.

Domänenkonflikte
Domänenkonflikte treten auf, wenn sich semantisch äquivalente Schemaattribute bezüglich der verwendeten Maßeinheiten unterscheiden. Demnach können Domänenkonflikte bei Kenngrößen, Klassifikationsstufen und (nicht-)dimensionalen Attributen auftreten.

Beispiel 3.4: Im MFUA-Schema werden in den Umsatz-Kenngrößen-Attributen die Werte in Euro ausgewiesen, wogegen im MFUB-Schema für die Darstellung der Umsätze US-Dollar verwendet wird. Analog erfolgt die Speicherung der Grundgebühren im entsprechenden nicht-dimensionalen Attribut im MFUA-Schema in Euro, im MFUB-Schema jedoch in US-Dollar. Darüber hinaus sind die beiden Dimensionen „Datum“ durch heterogene Hierarchien gekennzeichnet: Die Hierarchie der Dimension „Datum“ im MFUB-Schema enthält die Klassifikationsstufe „Quartal“, welche die zusätzlichen roll-up Beziehungen Monat $\mapsto$ Quartal und Quartal $\mapsto$ Jahr erfordert.

Schema-Instanz-Konflikte
Schema-Instanz-Konflikte treten auf, wenn in einem Schema der Kontext einer Kenngröße von einer Instanz eines dimensionalen Attributs bestimmt wird (Fakt-Kontext als Dimensi-

onsinstanz) und in anderen Schemata separate Kenngrößen für die unterschiedlichen Kontexte vorhanden sind (Dimensionsinstanzen als kontextualisierte Fakten).

Beispiel 3.5: Im MFUA-Schema existiert für Telefonie- und sonstige Umsätze je ein Kenngrößen-Attribut „Umsatz_Telefonie“ und „Umsatz_Sonstiges“, wogegen das MFUB-Schema den Kontext des Umsatzes (Telefonie-Umsatz oder sonstiger Umsatz) mit einer Instanz des dimensionalen Attributs „Umsatzkategorie“ beschreibt.

3.3.3.2 Konflikte auf Instanzebene

Die folgenden Tabellen visualisieren die Instanzen der logischen DW-Schemata (Star-Schemata) für MFUA und MFUB, wobei die erläuterten Heterogenitäten „Dimensionalität“ und „Schema-Instanz-Konflikt“ aufgelöst wurden. Mit Hilfe dieser Tabellen wird bei der Erläuterung der Konflikte auf Instanzebene auf das Rahmenbeispiel Bezug genommen.

mfuA:Verkauf

Datum	***svnr***	***Mobilnetz***	**Dauer**	**Umsatz_Telefonie**	**Umsatz_Sonstiges**
01.01.2006	1234010180	MobCom	58	11,60	1,00
30.06.2006	9876090875	HandyTel	130	14,20	4,00

mfuA:Datum

Datum	**Monat**	**Jahr**	**ALL**
01.01.2006	$\mapsto$ 01/06	$\mapsto$ 2006	$\mapsto$ all
30.06.2006	$\mapsto$ 06/06	$\mapsto$ 2006	$\mapsto$ all

mfuA:Kunde

svnr	**Name**	**Tarif**	**Bezeichnung**	**Grundgebühr**	**ALL**
1234010180	Mustermann	$\mapsto$ SparCom	SparCom Tarif	15	$\mapsto$ all
9876090875	Musterfrau	$\mapsto$ FlatTarif	FlatTarif Tarif	45	$\mapsto$ all

mfuA:Mobilnetz

Mobilnetz	**Bezeichnung**	**Einwahlgebühr**	**ALL**
MobCom	Mobile Communications	0,1	$\mapsto$ all
HandyTel	Handy Telephony Inc.	0,08	$\mapsto$ all
FiveCom	Five Communications Inc.	0,12	$\mapsto$ all

Tabelle 3.2: Rahmenbeispiel: DW-Instanzen von MFUA (Star-Schema)

mfuB:Verkauf

Datum	***Kunde***	***Mobilnetz***	**Werbeaktion**	**Umsatzkategorie**	**Gesprächsdauer**	**Umsatz**
01.01.2006	1234010180	MobCom	null	Umsatz_Telefonie	17	2,55
25.08.2006	6785041070	HandyTelInc	Sommer2006	Umsatz_Telefonie	50	6,20
25.08.2006	6785041070	HandyTelInc	Sommer2006	Umsatz_Sonstiges	0	3,50

mfuB:Verkauf (Heterogenitäten „Dimensionalität“ und „Schema-Instanz-Konflikt“ aufgelöst)

Datum	***Kunde***	***Mobilnetz***	**Gesprächsdauer**	**Umsatz_Telefonie**	**Umsatz_Sonstiges**
01.01.2006	1234010180	MobCom	17	2,55	0
25.08.2006	6785041070	HandyTelInc	50	6.20	3.50

mfuB:Datum

Datum	**Month**	**Quartal**	**Jahr**	**ALL**
01.01.2006	$\mapsto$ 01/06	$\mapsto$ 1/06	$\mapsto$ 2006	$\mapsto$ all
25.08.2006	$\mapsto$ 08/06	$\mapsto$ 3/06	$\mapsto$ 2006	$\mapsto$ all

mfuB:Kunde

Kunde	**Name**	**Altersgruppe**	**Tarif**	**TName**	**Grundgebühr**	**ALL**
1234010180	Mustermann	31-40	$\mapsto$ FullCom	FullCom Tarif	15	$\mapsto$ all
6785041070	Testperson	21-30	$\mapsto$ 2For0	2For0 Tarif	20	$\mapsto$ all

mfuB:Mobilnetz

Mobilnetz	**Bezeichnung**	**ALL**
MobCom	Mobile Communications	$\mapsto$ all
HandyTelInc	Handy Telephony Inc.	$\mapsto$ all
MobileCall	Mobile Calling Inc.	$\mapsto$ all
WirelessCom	Wireless Communications	$\mapsto$ all

Tabelle 3.3: Rahmenbeispiel: DW-Instanzen von MFUB (Star-Schema)

Überlappende/Disjunkte Fakten
Überlappende Fakten bezeichnen Fakten, welche identische Primärschlüssel besitzen. Dabei können zwei semantische Beziehungen unterschieden werden:

- **Identitätsbeziehung:** Die Fakten beschreiben dasselbe Objekt der Wirklichkeit. In diesem Zusammenhang ist zu beachten, dass nur eine der beiden durch die Primärschlüssel identifizierten Fakten wahr sein kann. Aus diesem Grund können hier auch keine Aggregatsfunktionen angewendet werden.
- **Kontextbeziehung:** Die Fakten beschreiben dasselbe Objekt der Wirklichkeit in einem unterschiedlichen Kontext bzw. unterschiedliche Objekte in ähnlichen Situationen. Daraus folgt, dass Fakten zusätzliche, versteckte Information enthalten können. Aggregatsfunktionen können grundsätzlich angewendet werden, jedoch werden in manchen Situationen nicht-sinnvolle Ergebnisse berechnet.

Beispiel 3.6: Die Fakttabellen von MFUA und MFUB (um Heterogenitäten bereinigt) speichern jeweils einen Fakt mit dem Primär-schlüssel {01.01.2006, 1234010180, MobCom}. In diesem Fall handelt es sich um eine Kontextbeziehung, da der Fakt jeweils aus der Sicht des jeweiligen Mobilfunkunternehmens beschrieben wird.

Für jene Tupel der Fakttabellen, welche unterschiedliche Primärschlüssel aufweisen, muss entschieden werden, wie bzw. ob diese im globalen Würfel bzw. im Ergebnis einer Abfrage enthalten sein sollen. Die Faktmengen können mit den bekannten SQL-Mengenoperatoren UNION, JOIN, MINUS, FULL OUTER JOIN, ... [KE04] integriert werden. Eine derartige Entscheidung muss auch für das Rahmenbeispiel getroffen werden.

Namenskonflikte
Auch auf Ebene der Dimensionsinstanzen können Namenskonflikte auftreten.

Beispiel 3.7: Die Dimensionstabelle „Mobilnetz“ enthält im MFUA-Schema eine Instanz „HandyTel“, im MFUB-Schema eine Instanz „HandyTelInc“; beide beschreiben jedoch dasselbe Objekt der Wirklichkeit (Synonym).

Überlappende/Disjunkte Dimensionsinstanzen
Analog zu überlappenden/disjunkten Fakten muss auch für überlappende/disjunkte Dimensionsinstanzen entschieden werden, wie bzw. ob diese in den globalen Würfel bzw. in das Ergebnis einer Abfrage integriert werden. Eine derartige Entscheidung muss auch für das Rahmenbeispiel getroffen werden.

Heterogene roll-up Beziehungen
Heterogene roll-up Beziehungen treten auf, wenn unterschiedliche Beziehungen zwischen Instanzen semantisch äquivalenter Klassifikationsstufen zweier oder mehrerer Dimensionen bestehen.

Beispiel 3.8: In der Dimensionstabelle „Kunde“ des MFUA-Schemas besteht eine roll-up Beziehung $1234010180 \mapsto$ SparCom wogegen im MFUB-Schema in der entsprechenden Tabelle eine roll-up Beziehung $1234010180 \mapsto$ FullCom besteht.

3.3.4 DBS-Heterogenitäten vs. DWS-Heterogenitäten

Dieser Abschnitt stellt die bei der Integration von DBS möglichen semantischen Heterogenitäten (vgl. Abschnitt 3.2.3) jenen, die bei der Integration von DWS auftreten können (vgl. Abschnitt 3.3.3), gegenüber.

Analog zur Integration von DBS können auch bei der Integration von DWS Namens- und Domänenkonflikte sowie Schema-Instanz-Konflikte auftreten. Die bei den semantischen Heterogenitäten zwischen DBS angeführten Konflikte aufgrund fehlender oder widersprüchlicher Daten entsprechen den Heterogenitäten aufgrund überlappender/disjunkter Fakten bzw. überlappender/disjunkter Dimensionsinstanzen. Bei den möglichen Heterogenitäten zwischen DWS wurden Konflikte aufgrund unterschiedlicher Datenrepräsentationen bzw. Datentypen nicht explizit angeführt. Jedoch können derartige Konflikte klarerweise auch die Integration von DWS erschweren.

Wie bereits eingangs erwähnt, treten bei der Integration von DWS zusätzliche Konflikte auf: Aufgrund des multidimensionalen Datenmodells bestehen auf Schemaebene Konflikte aufgrund unterschiedlicher Dimensionalität der Würfel, was für relationale DBS nicht gilt. Weitere zusätzliche Konflikte sind, auf Schemaebene, heterogene Aggregationshierarchien und, auf Instanzebene, heterogene roll-up Beziehungen.

3.3.5 Lösungsansätze

Analog zu den im Abschnitt 3.2.4 diskutierten Lösungsansätzen für die Auflösung von Heterogenitäten bei der Integration von DBS, muss auch bei der Integration von DWS zunächst eine Variante eines integrierten DWS ausgewählt werden. In einem FDWS müssen die Konflikte im Zuge der Erstellung des globalen DW-Schemas bzw. des globalen Würfels aufgelöst werden, in einem MDWS erfolgt die Konfliktbereinigung durch spezielle Abfragesprachen. Die im Abschnitt 3.2.4 angesprochenen Lösungsansätze für die Integration von DBS können zum Teil auch für die Integration von DWS herangezogen werden.

Auf dem Gebiet der Integration von DWS wurde bis jetzt vor allem die Problematik der Integration von Dimensionen untersucht: In [TP05] wird ein Werkzeug für die Integration von Dimensionen präsentiert, welches auf den in [CT05] und [CT04] präsentierten theoretischen Konzepten basiert. In [CT05] und [CT04] postulieren die Autoren das Konzept der Dimensions-Kompatibilität als Voraussetzung für die Integration von autonomen Data Marts. Auf Basis kompatibler Dimensionen können Data Mart-übergreifende Abfragen („drill-across“) definiert werden. Aufgrund der losen Kopplung ist das beschriebene System als MDWS zu charakterisieren. Ein ähnlicher Ansatz wird in [HGM05] vorgeschlagen, wo mit einem Graph-Modell Gemeinsamkeiten zwischen Dimensionen identifiziert werden können. In [CT05] wird zusätzlich ein eng-gekoppeltes System beschrieben, welches auf materialisierten Sichten für autonome Data Marts basiert. [BS08]

Die im Abschnitt 3.3.2 präsentierte Architektur eines FDWS gewährleistet dagegen eine ganzheitliche Integration von Fakten und Dimensionen in Form eines eng-gekoppelten Systems, welches transparenten Zugriff auf die Komponenten-DWS garantiert. Um Transparenz für den Benutzer sicherzustellen, ist es erforderlich die zwischen den zu integrierenden Würfeln bestehenden Heterogenitäten aufzulösen. Aus diesem Grund werden im Abschnitt 4.3 ausgewählte, für OLAP-Anwendungen geeignete Abfragesprachen hinsichtlich ihrer Einsetzbarkeit für die Integration von DWS analysiert. Es wird insbesondere beurteilt, inwieweit die erläuterten Heterogenitäten aufgelöst werden können. Im Kapitel 5 wird die multidimensionale Abfragesprache SQL-MDi vorgestellt, welche spezielle Konstrukte für die Auflösung der diskutierten Konflikte umfasst und somit die Integration von autonomen DWS umfassend unterstützt.

3.4 Zusammenfassung

Der Schwerpunkt dieses Kapitels lag auf der Problematik der Integration von DBS bzw. der Integration von DWS. Zu Beginn wurden zwei allgemeine Ansätze zur Datenintegration, nämlich „Local as view“ und „Global as view“ vorgestellt, welche sich im Wesentlichen hinsichtlich der Mapping-Definitionen unterscheiden. Die Problematik der DBS-Integration bzw. der DWS-Integration resultiert vor allem aus den zwischen den Komponentensystemen bestehenden Heterogenitäten.

Im Zuge der Erläuterung der DBS-Integration wurden verschiedene Typen von verteilen DBS, insbesondere MDBS und FDBS, kurz vorgestellt und eingeordnet. Aufgrund der Bedeutung für die Themenstellung dieser Diplomarbeit wurden FDBS vertieft behandelt, indem das Fünf-Ebenen-Schema-Architekturmodell für FDBS erläutert wurde. Ein auf diesem Modell basierendes FDBS schirmt den Benutzer von der Datenverteilung und von bestehenden Heterogenitäten ab. Abschließend wurden diverse Lösungsansätze für die Integration von DBS samt Literaturverweisen angeführt.

Analog zur Erläuterung der DBS-Integration wurden verschiedene Typen von verteilten DWS klassifiziert, wobei in diesem Kontext ein Architekturmodell für FDWS präsentiert wurde. Das vorgestellte Architekturmodell ermöglicht die Abschirmung des Benutzers von bestehenden Heterogenitäten. Im Anschluss wurde die Problematik der semantischen Heterogenitäten zwischen DWS behandelt, indem eine Klassifizierung der unterschiedlichen Heterogenitäten mit Hilfe des Rahmenbeispiels erfolgte. Im Vergleich zu den Heterogenitäten, welche bei der Integration von DBS auftreten, wird die Integration von DWS durch zusätzliche Heterogenitäten verkompliziert. Der Grund dafür liegt darin, dass das von DWS verwendete multidimensionale Datenmodell semantisch reichhaltiger ist als das relationale Datenmodell. Zuletzt wurde der Stand der Forschung auf dem Gebiet der Integration von DWS behandelt, indem entwickelte Lösungsansätze angesprochen und Literaturverweise angeführt wurden.

Teil II

Sprachen für die Integration von Data Warehouse-Systemen

Kapitel 4

Analyse von Abfragesprachen für die Integration von DBS/DWS

Die Problematik der Heterogenität bei der Integration von DBS führte zur Entwicklung von Abfragesprachen, welche durch die simultane Verarbeitung von Daten und Metadaten die Integration von heterogenen Schemata ermöglichen. Die Abfragesprachen sollen also die Integration von DBS erlauben, wo Daten einer Datenbank in einer anderen Datenbank als Metadaten bzw. Schemaattribute modelliert sind (vgl. Schema-Instanz-Konflikte im Abschnitt 3.3.3.1). Der Großteil dieser Abfragesprachen wurde für den Einsatz in einem MDBS konzipiert. Für die Integration von DWS, wo zusätzliche Heterogenitäten auftreten können, wurde kaum Forschungsarbeit geleistet. Das Ziel dieses Kapitels ist die Analyse ausgewählter Abfragesprachen hinsichtlich ihrer Eignung für den Einsatz in der Föderations-Schicht des im Abschnitt 3.3.2 präsentierten FDWS-Architekturmodells. Der Zweck der Verwendung einer Abfragesprache in der Föderations-Schicht des FDWS ist die Auflösung der bestehenden semantischen Heterogenitäten auf Schema- und Instanzebene.

Im Abschnitt 4.2 werden ausgewählte Abfragesprachen für MDBS kurz vorgestellt. Basierend auf der Annahme, dass das gegenständliche FDWS relationale Komponenten-DWS integriert (vgl. Abschnitt 1.2), wurde untersucht, ob diese Abfragesprachen für die Integration von solchen Systemen grundsätzlich geeignet sind. Für diesen Zweck wird im Abschnitt 4.1 ein Katalog von Kriterien zur Beurteilung der Abfragesprachen vorgestellt. Im Abschnitt 4.3 werden die Konzepte der für die Integration von DWS grundsätzlich geeigneten Abfragesprachen näher erläutert und anhand des Rahmenbeispiels demonstriert, inwieweit die im Abschnitt 3.3.3 beschriebenen Heterogenitäten aufgelöst werden können. Eine zusammenfassende Beurteilung der im Abschnitt 4.3 vorgestellten Abfragesprachen folgt im letzten Abschnitt 4.3.4.

4.1 Beurteilungskriterien

In diesem Abschnitt wird ein Katalog von Kriterien vorgestellt, anhand derer die Eignung der vorgestellten Abfragesprachen für den Einsatz in der Föderations-Schicht eines FDWS beurteilt wird. Jene Abfragesprachen, welche die definierten Kriterien erfüllen, werden im Abschnitt 4.3

näher vorgestellt. Es wird insbesondere gezeigt, inwieweit die im Abschnitt 3.3.3 beschriebenen Heterogenitäten eliminiert werden können. Der Katalog umfasst folgende Kriterien:

1. **SQL-Kompatibilität (SQL)**
   Im Abschnitt 1.2 wurde angenommen, dass die im FDWS zu integrierenden Komponenten-DWS ein relationales Speichermodell (vgl. Abschnitt 2.4.2) implementieren. Aus diesem Grund ist es möglich die SQL-Fähigkeiten der Komponenten-DWS für die Integrationsaufgaben zu nutzen. Die in Frage kommenden Abfragesprachen haben daher Kompatibilität mit SQL zu gewährleisten, um die in der relationalen Algebra [KE04] definierten Operationen, wie Selektion, Projektion oder Umbenennung, verwenden zu können. Die Mächtigkeit der relationalen Algebra reicht aus, um den Großteil der im Abschnitt 3.3.3 beschriebenen Heterogenitäten zu eliminieren.
2. **Aggregatsfunktionen (AGG)**
   Die in Frage kommenden Abfragesprachen haben zumindest die in SQL definierten, „primitiven“ Aggregatsfunktionen MIN, MAX, COUNT, SUM und AVG zu unterstützen. Da die Berechnung von Aggregationen in einem DWS unabdingbar ist, sind Sprachen mit erweiterten Funktionalitäten zur Berechnung von Aggregationen als mächtiger einzustufen.
3. **Daten-Metadaten-Transformationen (DMT)**
   Zur Auflösung von Schema-Instanz-Konflikten (vgl. Abschnitt 3.3.3.1), wo Daten eines Schemas in einem anderen Schema als Metadaten modelliert sind, haben die in Frage kommenden Abfragesprachen die simultane Verarbeitung von Daten und Metadaten, d.h. die Transformation von Daten zu Metadaten und umgekehrt, zu unterstützen. Auf Basis dieser Eigenschaft können komplexe Schema-Restrukturierungen vorgenommen werden.
4. **Dynamische Output-Schemata (DOS)**
   Die Fähigkeit zur Erzeugung dynamischer Output-Schemata beschreibt, inwieweit eine unterschiedliche Anzahl von Relationen bzw. eine unterschiedliche Anzahl von Attributen einer Relation erzeugt werden kann [WR05]. Die in Frage kommenden Abfragesprachen haben dieses Kriterium zu erfüllen, um nicht-benötigte Relationen und Attribute im globalen Schema ausblenden zu können.
5. **Multidimensionales Datenmodell (MD)**
   Da Data Warehouses auf einem multidimensionalen Datenmodell basieren, sollten auch Abfragesprachen ein solches unterstützen. Da jedoch, soweit bekannt, nur wenige multidimensionale Abfragesprachen entwickelt wurden und ein relationales Speichermodell in den Komponenten-DWS angenommen wird, ist die Erfüllung dieses Kriteriums nicht zwingend erforderlich, solange alle übrigen erfüllt sind. Implementiert jedoch eine Abfragesprache ein multidimensionales Datenmodell, so müssen die übrigen Kriterien nicht erfüllt sein. Es wird angenommen, dass die dadurch mögliche Berücksichtigung der im Abschnitt 2.3 vorgestellten Konstrukte des multidimensionalen Datenmodells eine elegante Auflösung der DW-spezifischen Heterogenitäten (vgl. Abschnitt 3.3.4) ermöglicht. Darüber hinaus würde ein konzeptuelles, multidimensionales Datenmodell auch die Integration von MOLAP-Systemen (vgl. Abschnitt 2.4.2) unterstützen.

Eine für die Integration von DWS, eingesetzt in der Föderations-Schicht eines FDWS, in Frage kommende Abfragesprache muss entweder die Kriterien 1 - 4 und/oder das Kriterium 5 erfüllen.

4.2 Überblick über Abfragesprachen für MDBS

In diesem Abschnitt werden ausgewählte Abfragesprachen für MDBS kurz vorgestellt. Mit Hilfe des im Abschnitt 4.1 präsentierten Kriterienkatalogs wird beurteilt, ob die Abfragesprachen für die Integration von DWS, eingesetzt in der Föderations-Schicht eines FDWS, geeignet sind. Die Abfragesprachen werden chronologisch, nach dem Jahr ihrer erstmaligen Veröffentlichung, behandelt.

4.2.1 HiLog

HiLog ist eine auf Prolog basierende, logische Programmiersprache, welche u.a. zur Datenbankprogrammierung eingesetzt werden kann. Durch die Möglichkeit der simultanen Verarbeitung von Daten und Metadaten in einer Abfrage eignet sich HiLog für die Integration in einem MDBS. Diese Mächtigkeit erreicht HiLog dadurch, indem die Prädikatenlogik erster Ordnung, die auch SQL und der relationalen Algebra (RA) zugrunde liegt, um Fähigkeiten höherer Ordnung erweitert wird. Diese Fähigkeiten unterstützen die simultane Verarbeitung von Daten und Metadaten. [CKW93] [WR05] Einige der im Anschluss vorgestellten Abfragesprachen verfolgen ebenfalls einen derartigen Ansatz.

Die Mächtigkeit von HiLog für die Integration von DBS resultiert aus der Unterstützung von Daten-Metadaten-Transformationen. Eine Implementierung von HiLog wird jedoch aufgrund der logischen Syntax, welche Rekursionen erlaubt, und der somit fehlenden SQL-Kompatibilität erschwert. Darüber hinaus sind keine Aggregatsfunktionen und dynamischen Output-Schemata möglich. Aufgrund des Fokus auf Datenbanken wird auch kein multidimensionales Datenmodell unterstützt.

Kriterien	
SQL-Kompatibilität	
Aggregatsfunktionen	
Daten-Metadaten-Transformationen	✓
Dynamische Output-Schemata	
Multidimensionales Datenmodell	

Tabelle 4.1: HiLog: Beurteilung

4.2.2 MSQL

MSQL (Multidatabase SQL) und die zugrunde liegende multirelationale Algebra (MRA) erweitern SQL bzw. die RA um die Fähigkeit, Abfragen auf eine Menge von Relationen (Multirelation), möglicherweise in unterschiedlichen Datenbanken, zu formulieren. Eine multirelationale Abfrage definiert eine Ziel-Multirelation aus einer Quell-Multirelation mit Hilfe von multirelationalen Operatoren, welche auf ihren relationalen Entsprechungen basieren (z.B. MPROJECT vs. PROJECT, MSELECT vs. SELECT, MJOIN vs. JOIN, etc.). Die Verarbeitung der Abfragen erfolgt durch Übersetzung der MSQL-Anweisungen in äquivalente MRA-Anweisungen, welche wiederum in eine Menge von RA-Anweisungen transformiert werden. [GLRS93]

Durch die Transformation von MRA-Anweisungen in eine Menge von RA-Anweisungen ist die SQL-Kompatibilität gewährleistet und es werden auch die primitiven Aggregatsfunktionen unterstützt. Jedoch erlaubt MSQL keine Daten-Metadaten-Transformationen und dynamischen Output-Schemata [WR05]. Aufgrund des Fokus auf Datenbanken wird auch kein multidimensionales Datenmodell unterstützt.

Kriterien	
SQL-Kompatibilität	✓
Aggregatsfunktionen	✓
Daten-Metadaten-Transformationen	
Dynamische Output-Schemata	
Multidimensionales Datenmodell	

Tabelle 4.2: MSQL: Beurteilung

In [MR95] wird eine Erweiterung von MSQL zur Auflösung von Schema-Instanz-Konflikten vorgestellt. Mit Hilfe von globalen Attributen werden Typ-kompatible Ausdrücke auf lokale Attribute formuliert. Die Realisierung erfolgt durch eine globale Klasse, welche alle globalen Attribute definiert und deren Sub-Klassen Attributdefinitionen für die einzelnen Komponenten-Schemata enthalten. Die Transformation von lokalen Attributen auf globale Attribute erfolgt durch Definition von Mappings (vgl. Definition 3.5). Mit dieser Erweiterung unterstützt MSQL zusätzlich Daten-Metadaten-Transformationen.

Kriterien	
SQL-Kompatibilität	✓
Aggregatsfunktionen	✓
Daten-Metadaten-Transformationen	✓
Dynamische Output-Schemata	
Multidimensionales Datenmodell	

Tabelle 4.3: Erweitertes MSQL: Beurteilung

4.2.3 IDL

IDL (Interoperable Database Language) besitzt, wie HiLog (vgl. Abschnitt 4.2.1), eine logische Syntax mit Fähigkeiten höherer Ordnung [KLK91], um innerhalb einer Anweisung sowohl Daten als auch Metadaten verarbeiten zu können. Im Gegensatz zu HiLog sind jedoch keine Rekursionen möglich [WR05], wodurch Effizienzprobleme bei der Abfrageverarbeitung vermieden werden. Das zugrunde liegende Datenmodell bedient sich geschachtelter Objektstrukturen zur Repräsentation von Metadaten: „das Universum ist ein Tupel von Datenbanken, wobei jede Datenbank wiederum ein Tupel von Relationen darstellt, welche jeweils eine Menge von Tupel darstellen und jedes Tupel ist wiederum ein Tupel von Objekten“. [KLK91]

Die Nutzung von Fähigkeiten höherer Ordnung erlaubt die Unterstützung von Daten-Metadaten-Transformationen. Jedoch ist das Kriterium der SQL-Kompatibilität aufgrund der logischen Syntax nicht erfüllt. Darüber hinaus unterstützt IDL, soweit bekannt, keine Aggregatsfunktionen und dynamischen Output-Schemata. Aufgrund des Fokus auf Datenbanken wird auch kein multidimensionales Datenmodell implementiert.

Kriterien	
SQL-Kompatibilität	
Aggregatsfunktionen	
Daten-Metadaten-Transformationen	✓
Dynamische Output-Schemata	
Multidimensionales Datenmodell	

Tabelle 4.4: IDL: Beurteilung

4.2.4 SchemaLog

SchemaLog ist, wie HiLog (vgl. Abschnitt 4.2.1), eine logische Programmiersprache, welche Fähigkeiten höherer Ordnung nutzt, um Daten und Metadaten simultan zu verarbeiten, wodurch Schema-Transformationen möglich sind [GLS+97]. In SchemaLog können Variablen für eine Menge von Objekten (z.B. Relationen oder Attribute) deklariert werden. Durch diese Ausdrucksstärke und die Möglichkeit Daten und Metadaten simultan zu verarbeiten, können globale Sichten zur Integration von heterogenen Schemata formuliert werden. Im Falle von Schema-Modifikationen („Schema Evolution“ [LSS93]) können Mappings zwischen altem und neuem Schema definiert werden um für die Benutzer „Evolutionstransparenz“ zu gewährleisten. [LSS93]

Darüber hinaus erweist sich SchemaLog bezüglich der Aggregationsfähigkeiten als mächtig: Es unterstützt, wie SQL, die vertikale Aggregation, zusätzlich horizontale Aggregation (über mehrere Attribute) und globale Aggregation (über Blöcke anderer Relationen) [LSS97]. Da SchemaLog die simultane Verarbeitung von Daten und Metadaten ermöglicht, können Daten-Metadaten-Transformationen für die Integration heterogener Schemata genutzt werden. Durch die Definition von Variablen, welche eine Menge von Objekten repräsentieren, können dynami-

sche Output-Schemata erzeugt werden. Die logische Syntax und die dadurch möglichen, ineffizienten Rekursionen verhindern jedoch die Erfüllung des Kriteriums der SQL-Kompatibilität [WR05]. Aufgrund des Fokus auf Datenbanken wird auch kein multidimensionales Datenmodell implementiert.

Kriterien	
SQL-Kompatibilität	
Aggregatsfunktionen	✓
Daten-Metadaten-Transformationen	✓
Dynamische Output-Schemata	✓
Multidimensionales Datenmodell	

Tabelle 4.5: SchemaLog: Beurteilung

4.2.5 UDM

Die Kernidee des UDM (Uniform Data Model) ist die Gleichbehandlung von Daten und Metadaten als Informationsobjekte, welche als „uniform databases“ modelliert werden. Jedes Daten- bzw. Metadatenobjekt wird in einer binären Relation kodiert, welche aus einer Spalte „Type“ und einer Spalte „RelationScheme“ (für Metadatenobjekte) bzw. „RelationTupel“ (für Datenobjekte) besteht. Die auf diesem Datenmodell basierende UDM-Algebra und das UDM-Kalkül können Daten und Metadaten simultan verarbeiten. [JMG95]

Da UDM eine Erweiterung des relationalen Datenmodells darstellt, ist die Kompatibilität mit SQL gewährleistet. Aufgrund der Fähigkeit Daten und Metadaten simultan zu verarbeiten sind darüber hinaus Daten-Metadaten-Transformationen möglich. Jedoch werden, soweit bekannt, Aggregatsfunktionen und dynamische Output-Schemata in der UDM-Algebra und im UDM-Kalkül nicht berücksichtigt. Aufgrund des Fokus auf Datenbanken wird auch kein multidimensionales Datenmodell unterstützt.

Kriterien	
SQL-Kompatibilität	✓
Aggregatsfunktionen	
Daten-Metadaten-Transformationen	✓
Dynamische Output-Schemata	
Multidimensionales Datenmodell	

Tabelle 4.6: UDM: Beurteilung

4.2.6 RSQL, RRA und SISQL

Diese Abfragesprachen nutzen die Eigenschaft des Programmierkonstrukts „Reflection“, mit welchem Programme selbst andere Programme erzeugen, manipulieren und evaluieren können. Auf Datenbanken bezogen bedeutet das, dass Abfragen dynamisch weitere Abfragen erzeugen können. Das in [DJGM] vorgestellte RSQL (Reflective SQL) erweitert SQL um Operatoren zum Codieren beliebiger SQL-Anweisungen in sogenannte „program tables (p-tables)“ (`REIFY`) sowie zum Decodieren und Auswerten der p-tables (`EVAL`). Die Kernidee hinter dieser Technik ist, dass p-tables wiederum Ziel von Abfragen sein können. Das Ziel bei der Verwendung von RSQL ist nicht primär die Schemaintegration durch Transformation heterogener Schemata, sondern die Formulierung von schemaunabhängigen Abfragen. Reflection ermöglicht somit das Formulieren von Abfragen unabhängig vom Schemawissen und vermeidet eine Neuformulierung von Abfragen bei Schemaänderungen. Ähnliche Ansätze verfolgen RRA (Reflective Relational Algebra) [dBGV96] und SISQL (Schema-Independent SQL) [MV00], wobei SISQL die Definition von Platzhaltern für Relationen und Attribute erlaubt um dynamisch RSQL-Anweisungen mit Hilfe eines „Data Dictionary“ zu erzeugen.

Da RSQL und SISQL SQL-Erweiterungen darstellen, ist das Kriterium der SQL-Kompatibilität erfüllt. Aufgrund der nicht-Fokussierung auf Schemaintegration werden Daten-Metadaten-Transformationen nicht unterstützt. Darüber hinaus werden, soweit bekannt, keine Aggregatsfunktionen und dynamischen Output-Schemata [WR05] berücksichtigt. Aufgrund des Fokus auf Datenbanken wird auch kein multidimensionales Datenmodell unterstützt.

Kriterien	
SQL-Kompatibilität	✓
Aggregatsfunktionen	
Daten-Metadaten-Transformationen	
Dynamische Output-Schemata	
Multidimensionales Datenmodell	

Tabelle 4.7: RSQL, RRA, SISQL: Beurteilung

4.2.7 Tabular Algebra

In [GLS96] wird die Tabular Algebra als eine Sprache für die Abfrage und Restrukturierung von tabellarischen Daten charakterisiert. In diesem Zusammenhang wird angeführt, dass unterschiedlichste Abfragesprachen einfach in die Tabular Algebra eingebettet werden können. Daher kann die Tabular Algebra als formale Basis für Abfragesprachen betrachtet werden. So verwenden z.B. SchemaLog (vgl. Abschnitt 4.2.4), SchemaSQL (vgl. Abschnitt 4.2.8), nD-SQL (vgl. Abschnitt 4.2.10) und CQL (vgl. Abschnitt 4.2.9) theoretische Konzepte der Tabular Algebra. Das zugrunde liegende Datenmodell bietet ein breiteres Spektrum an tabellarischen Darstellungsformen als das relationale Datenmodell, da neben Spaltenattributen auch Zeilenattribute zulässig sind, welche sowohl Daten als auch Metadaten enthalten können. Darüber hinaus wird die simultane Verarbeitung von Daten und Metadaten durch eine Reihe von spe-

ziellen Algebra-Operatoren (`GROUP`, `MERGE`, `SPLIT`, `COLLAPSE`) sichergestellt, wodurch die meisten in der Praxis auftretenden Schematransformationen durchgeführt werden können. Aufgrund ihrer Analogie zur Tabellenkalkulation ist die Tabular Algebra als theoretische Grundlage für OLAP geeignet.

Da das Hauptaugenmerk der Tabular Algebra auf der Transformation von Schemata liegt und keine Implementierungsvorschrift vorliegt, kann diese als theoretische Grundlage für die Entwicklung einer Abfragesprache für die Integration von DBS bzw. DWS herangezogen werden. Die Erfüllung der Kriterien des Katalogs hängt somit von der verwendeten Implementierungssprache ab.

4.2.8 SchemaSQL

SchemaSQL ist eine Erweiterung von SQL und unterstützt die simultane Verarbeitung von Daten und Metadaten, wodurch die Integration von heterogenen Schemata ermöglicht wird. Dies wird erreicht, indem SchemaSQL, neben dem aus SQL bekannten Tupel-Variablentyp, zusätzliche Variablentypen implementiert um die Abfrage von Metadaten, wie Datenbanknamen, Relationennamen oder Attributnamen, zu unterstützen. Neben der bekannten vertikalen Aggregation, erlaubt SchemaSQL, wie SchemaLog (vgl. Abschnitt 4.2.4), horizontale Aggregation sowie Aggregation über Blöcke. [LSS01]

Da SchemaSQL eine SQL-Erweiterung darstellt, ist das Kriterium der SQL-Kompatibilität erfüllt. Weiters erweist sich SchemaSQL bezüglich der Aggregatsfunktionen als mächtig, da neben der aus SQL bekannten vertikalen Aggregation weiterreichende Funktionalitäten zur Verfügung stehen. Darüber hinaus wird die simultane Verarbeitung von Daten und Metadaten unterstützt, wodurch Daten-Metadaten-Transformationen ermöglicht werden. Durch Verwendung der erwähnten zusätzlichen Variablentypen in einer `CREATE VIEW` Anweisung können dynamische Output-Schemata erzeugt werden. Aufgrund des Fokus auf Datenbanken unterstützt SchemaSQL jedoch kein multidimensionales Datenmodell.

Kriterien	
SQL-Kompatibilität	✓
Aggregatsfunktionen	✓
Daten-Metadaten-Transformationen	✓
Dynamische Output-Schemata	✓
Multidimensionales Datenmodell	

Tabelle 4.8: SchemaSQL: Beurteilung

4.2.9 CQL

CQL (Cube-Query-Language) erlaubt die Formulierung von multidimensionalen Abfragen, welche in zwei Schritten, nämlich „Berechnung“ und „Präsentation“, verarbeitet werden. Durch diese Trennung ist es möglich, auf Basis dieser Eigenschaft, Anwendungen zu schreiben, wo ein Server CQL-Abfragen berechnet und die Clients die Ergebnispräsentation übernehmen. Als einzige der untersuchten Abfragesprachen implementiert CQL ein konzeptuelles, multidimensionales Datenmodell (CROSS-DB), welches die im Abschnitt 2.3 vorgestellten Konstrukte kennt, wodurch CQL unabhängig vom logischen DW-Modell (vgl. Abschnitt 2.4.2) eingesetzt werden kann. Basierend auf dem CROSS-DB-Modell, erlaubt CQL u.a. detaillierte Analysen aufgrund definierter Eigenschaften („features“) und geschachtelte Sub-Abfragen, welche Sub-Würfel repräsentieren, die wiederum zu einem Endergebnis zusammengefügt werden können. Basierend auf der Tabular Algebra (vgl. Abschnitt 4.2.7), bietet CQL verschiedenste Möglichkeiten zur Präsentation der Abfrageergebnisse. [BL97]

CQL wurde für OLAP-Anwendungen konzipiert, weshalb diese Sprache ein multidimensionales Datenmodell implementiert und mächtige Aggregationsfähigkeiten bietet. Trotz der Ähnlichkeit bezüglich der Syntax, stellt CQL keine SQL-Erweiterung dar und ist somit nicht SQL-kompatibel. Da CQL nicht für Integrationsaufgaben entwickelt wurde, werden keine Daten-Metadaten-Tranformationen und dynamischen Output-Schemata unterstützt.

Kriterien	
SQL-Kompatibilität	
Aggregatsfunktionen	✓
Daten-Metadaten-Transformationen	
Dynamische Output-Schemata	
Multidimensionales Datenmodell	✓

Tabelle 4.9: CQL: Beurteilung

4.2.10 nD-SQL

In [GL98] wird nD-SQL als eine multidimensionale Abfragesprache für die Integration von heterogenen, relationalen Schemata sowie Data Marts beschrieben. nD-SQL unterstützt die simultane Verarbeitung von Daten und Metadaten in einer Abfrage, indem Variablen für die Abfrage von Metadaten, z.B. für Datenbanknamen, Relationennamen, etc., deklariert werden können. Darüber hinaus erlaubt nD-SQL beliebige Kombinationen von GROUP-BYs zur Berechnung von Aggregationen. [GL98]

nD-SQL erfüllt das Kriterium der SQL-Kompatibilität [GL98]. Durch die Unterstützung von beliebigen GROUP-BY-Kombinationen erweist sich nD-SQL bezüglich der Aggregatsfunktionen als sehr mächtig. Weiters sind Daten-Metadaten-Transformationen durch simultane Verarbeitung von Daten und Metadaten sowie dynamische Output-Schemata durch Verwendung von Variablen für Metadaten, in Kombination mit speziellen Schlüsselwörtern, im `SELECT` Ab-

schnitt einer Abfrage möglich. Abschließend sei noch erwähnt, dass die Charakterisierung von nD-SQL als multidimensionale Abfragesprache irreführend ist, da diese auf dem relationalen Datenmodell basiert und kein multidimensionales Datenmodell unterstützt.

Kriterien	
SQL-Kompatibilität	✓
Aggregatsfunktionen	✓
Daten-Metadaten-Transformationen	✓
Dynamische Output-Schemata	✓
Multidimensionales Datenmodell	

Tabelle 4.10: nd-SQL: Beurteilung

4.2.11 MD-SQL, MQL/MA und FISQL/FIRA

MD-SQL (Meta-Data SQL) hat ihre Wurzeln in SchemaSQL (vgl. Abschnitt 4.2.8) und erlaubt ebenso die simultane Verarbeitung von Daten und Metadaten. Im Unterschied zu SchemaSQL wird eine MD-SQL-Abfrage in eine Menge von atomaren, relationalen Anweisungen transformiert. MD-SQL erweitert SQL um zwei Konstrukte: Mit dem Schlüsselwort `AS` im `SELECT` Abschnitt können Spalten und Relationen umbenannt werden, wobei mittels Schachtelungen spezifiziert werden kann, ob die Abfrage eine Relation oder eine Datenbank (Menge von Relationen) als Ergebnis liefert. Das Schlüsselwort `ALL` im `SELECT` Abschnitt spezifiziert eine Gruppe von zu selektierenden Attributen („generalized projection“); im `FROM` Abschnitt bewirkt es eine Verbund-Operation auf alle Relationen einer Sub-Abfrage und liefert eine Relation als Endergebnis („generalized join“). Mit letzterem Konstrukt können Output-Schemata ohne Definition einer Sicht, wie es in SchemaSQL erforderlich ist, dynamisch erzeugt werden. [RGW99]

MQL (Meta-Query Language) und die zugrunde liegende MA (Meta-Algebra) sowie FISQL (Federated Interoperable Structured Query Language) und die zugrunde liegende FIRA (Federated Interoperable Relational Algebra) stellen Weiterentwicklungen von MD-SQL und SchemaSQL dar. [WG01] [WR05]

MD-SQL, MQL und FISQL/FIRA sind SQL-kompatible Abfragesprachen, welche Daten-Metadaten-Transformationen sowie dynamische Output-Schemata unterstützen. Jedoch werden, soweit bekannt, keine Aggregatsfunktionen unterstützt. Aufgrund des Fokus auf Datenbanken wird auch kein multidimensionales Datenmodell implementiert.

Kriterien	
SQL-Kompatibilität	✓
Aggregatsfunktionen	
Daten-Metadaten-Transformationen	✓
Dynamische Output-Schemata	✓
Multidimensionales Datenmodell	

Tabelle 4.11: MD-SQL, MQL, FISQL/FIRA: Beurteilung

4.2.12 SQL_M

SQL_M ist keine Abfragesprache für die Integration von DBS bzw. DWS, sondern ein multidimensionales Datenmodell, welches die Handhabung von irregulären Dimensionshierarchien in Data Warehouses erlaubt. Irreguläre Dimensionshierarchien bestehen, wenn Hierarchien „nicht-strikt“ sind, d.h. wenn Klassifikationsstufen niedrigerer Granularität mehrere Vaterelemente höherer Granularität haben können, oder „nicht-deckend“ sind, d.h. wenn Klassifikationsstufen beim roll-up übersprungen werden können. Die Problematik von irregulären Dimensionshierarchien besteht darin, dass sie den Grundsatz der „summarizability“ von OLAP-Daten verletzten [HGM05]. Das bedeutet, dass der Benutzer möglicherweise falsche Ergebnisse erhält, wenn er Zwischenresultate für die Berechnung von Ergebnissen höherer Granularität verwendet. Um „summarizability“ zu gewährleisten, bietet das SQL_M-Datenmodell die Möglichkeit der Definition von Aggregatstypen für jede Kombination von Kenngröße und Dimension, wodurch die Verwendung von bestimmten Aggregatsfunktionen, z.B. SUM, verboten werden kann. [PRP02] Die Problematik von irregulären Dimensionshierarchien wird auch in [HGM05] detailliert behandelt.

Da SQL_M keine Abfragesprache, sondern ein multidimensionales Datenmodell darstellt, wird es nicht mit dem Kriterienkatalog hinsichtlich der Einsetzbarkeit für die Integration von DWS beurteilt. Vielmehr kann SQL_M, wie die Tabular Algebra (vgl. Abschnitt 4.2.7), als Basis für die Entwicklung einer multidimensionalen Abfragesprache zur Integration von DWS dienen.

4.2.13 Beurteilung

Tabelle 4.12 fasst die Bewertungen der vorgestellten Abfragesprachen hinsichtlich der im Abschnitt 4.1 definierten Kriterien zusammen.

Abfragesprache	**SQL**	**AGG**	**DMT**	**DOS**	**MD**
HiLog			✓		
MSQL	✓	✓			
Erweitertes MSQL	✓	✓	✓		
IDL			✓		
SchemaLog		✓	✓	✓	
UDM	✓		✓		
RSQL, RRA, SISQL	✓				
SchemaSQL	✓	✓	✓	✓	
CQL		✓			✓
nD-SQL	✓	✓	✓	✓	
MD-SQL, MQL/MA, FISQL/FIRA	✓		✓	✓	

Tabelle 4.12: Beurteilung der Abfragesprachen

Wie aus der Analyse hervorgeht, wurde die Problematik der Schema-Instanz-Konflikte (vgl. Abschnitt 3.3.3.1) von der Forschung erkannt. Die Mehrzahl der untersuchten Abfragespra-

chen unterstützt deshalb Daten-Metadaten-Transformationen um Schema-Restrukturierungen zu ermöglichen. Da die meisten Abfragesprachen für die Integration von relationalen Datenbanken entwickelt wurden, ist, außer bei den logischen Sprachen sowie der multidimensionalen Sprache CQL, die Kompatibilität mit SQL gewährleistet. Aus dem selben Grund dürften Aggregatsfunktionen, welche in Data Warehouses besondere Bedeutung haben, weniger oft berücksichtigt worden sein. Die Fähigkeit zur Generierung dynamischer Output-Schemata besitzen nur vier der untersuchten Abfragesprachen.

Soweit bekannt, ist CQL die einzige, universell einsetzbare Abfragesprache, welche auf einem konzeptuellen, multidimensionalen Datenmodell basiert, wodurch CQL unabhängig vom logischen DW-Modell (vgl. Abschnitt 2.4.2) verwendet werden kann. Jedoch wurde CQL nicht für die Unterstützung der Integration von DWS entwickelt. Aufgrund der Tatsache, dass, soweit bekannt, noch keine Abfragesprache entwickelt wurde, welche die Integration von heterogenen DW-Schemata bzw. Würfeln unterstützt, ist auf diesem Gebiet ein Forschungsdefizit zu konstatieren.

Die Abfragesprachen SchemaSQL und nD-SQL sind die einzigen Abfragesprachen, welche die Kriterien SQL, AGG, DMT und DOS erfüllen und somit für die Integration von DWS, eingesetzt in der Föderations-Schicht eines FDWS, in Frage kommen. SchemaSQL und nD-SQL werden in den Abschnitten 4.3.1 bzw. 4.3.2 detaillierter behandelt; es wird insbesondere gezeigt ob bzw. wie die im Abschnitt 3.3.3 beschriebenen Heterogenitäten aufgelöst werden können. Da CQL als einzige Abfragesprache das Kriterium MD erfüllt, werden deren theoretische Konzepte sowie deren Eignung für die Integration von DWS im Abschnitt 4.3.3 näher beschrieben.

4.3 Analyse ausgewählter Abfragesprachen

In diesem Abschnitt werden die Konzepte der für die Integration von DWS, eingesetzt in der Föderations-Schicht eines FDWS, in Frage kommenden Abfragesprachen näher vorgestellt sowie ihre Eignung für die Auflösung der im Abschnitt 3.3.3 diskutierten Heterogenitäten beurteilt. Dabei wird auf die konzeptuellen DW-Schemata der Abbildung 3.4 sowie auf die DW-Instanzen der Tabellen 3.2 und 3.3 Bezug genommen. Gegenstand dieses Abschnitts sind die Abfragesprachen SchemaSQL, nD-SQL und CQL. Der neu entwickelten multidimensionalen Abfragesprache zur Integration von DWS, SQL-MDi, wird Kapitel 5 gewidmet.

4.3.1 SchemaSQL

SchemaSQL [LSS01] [LSS99] ist eine Erweiterung von SQL und erlaubt Abfragen in einem MDBS. Die wichtigste Erweiterung gegenüber SQL ist die Möglichkeit der simultanen Verarbeitung von Daten und Metadaten bzw. der Transformation von Daten zu Metadaten und umgekehrt, wodurch die Integration von heterogenen Schemata ermöglicht wird. Darüber hinaus bietet SchemaSQL erweiterte Aggregationsfähigkeiten, indem neben der aus SQL bekann-

ten vertikalen Aggregation auch horizontale Aggregation sowie Aggregation über Blöcke unterstützt wird. Unter horizontaler Aggregation versteht man die Berechnung von Aggregationen für eine Menge von Werten mehrerer Attribute einer Relation; Aggregation über Blöcke bezeichnet die Berechnung von Aggregationen für eine Menge von Werten mehrerer Attribute aus unterschiedlichen Relationen [LSS01].

4.3.1.1 Theoretische Konzepte

SchemaSQL unterstützt neben dem aus SQL bekannten Tupel-Variablentyp zusätzliche Typen, um die Abfrage von Metadaten und komplexe Schema-Restrukturierungen zu ermöglichen. SchemaSQL erlaubt die Deklaration von Variablen, welche über die folgenden fünf Mengen iterieren können:

1. `db-name` Variable über die Menge der Namen der Datenbanken einer Föderation (Syntaxelement: `->`)
2. `rel-name` Variable über die Menge der Namen der Relationen einer Datenbank `db` (Syntaxelement: `db->`)
3. `attr-name` Variable über die Menge der Namen der Attribute einer Relation `rel` einer Datenbank `db` (Syntaxelement: `db::rel->`)
4. `tuple` Variable über die Menge der Tupel einer Relation `rel` einer Datenbank `db` (Syntaxelement: `db::rel`)
5. `domain` Variable über die Menge der Werte einer einem Attribut `attr` zugeordneten Spalte einer Relation `rel` einer Datenbank `db` (Syntaxelement: `db::rel.attr`)

Die Deklaration von Variablen folgt der SQL-Syntax, nämlich `<range> <var>`, wobei `<range>` den Bereich, also eine der fünf Mengen angibt, und `<var>` einen Namen. Ein wichtiger Unterschied gegenüber SQL ist, dass die spezifizierten Bereiche geschachtelt sein können, da SchemaSQL unterschiedliche Variablentypen unterstützt (in SQL sind nur Tupel-Variablen erlaubt!). Es ist also möglich eine Variable X zu deklarieren, welche über die Relationen einer Datenbank D iteriert und eine Variable T, welche über die Tupel der durch X identifizierten Relation iteriert.

Nachdem nun in diesem Abschnitt die Erweiterungen gegenüber SQL, nämlich die zusätzlichen Variablentypen, erläutert wurden, wird im folgenden Abschnitt untersucht, inwieweit diese Erweiterungen zur Auflösung der im Abschnitt 3.3.3 diskutierten Heterogenitäten geeignet sind. Im Exkurs des Abschnitts 4.3.1.3 wird die erweiterte Aggregationsfähigkeit „horizontale Aggregation“ anhand einer Beispiel-Abfrage veranschaulicht.

4.3.1.2 Anwendung für die Integration von DWS

Für die Beurteilung der Eignung von SchemaSQL für die Integration von DWS wird jene im Abschnitt 3.3.3 verwendete Kategorisierung der verschiedenen Heterogenitäten sowie das Rahmenbeispiel der Abbildung 3.4 bzw. der Tabellen 3.2 und 3.3 herangezogen. Für jene Heterogenitäten, die auch mit SQL aufgelöst werden können, werden entsprechende SQL-Abfragen formuliert. SchemaSQL ist kompatibel mit SQL, da eine SchemaSQL-Abfrage in eine Menge von SQL-Abfragen transformiert wird [LSS01].

Konflikte auf Schemaebene

Im Folgenden wird demonstriert, wie Konflikte auf Schemaebene mit (Schema)SQL-Abfragen aufgelöst werden können.

Dimensionalität
Heterogenitäten aufgrund unterschiedlicher Dimensionalität können mit SQL-Abfragen aufgelöst werden, indem die dimensionalen Attribute jener Dimensionen einer Fakttabelle, welche nicht in allen zu integrierenden Schemata existieren, mit einer Projektion auf die Fakttabelle ausgeblendet werden.

Beispiel 4.1: Das im MFUB-Schema zusätzlich vorhandene dimensionale Attribut „Werbeaktion“ wird mit der SQL-Abfrage des Listing 4.1 in der Fakttabelle ausgeblendet:

```
1 SELECT v.Datum, v.Kunde, v.Mobilnetz, v.Umsatzkategorie, v.Gesprächsdauer, v.Umsatz
2 FROM mfuB:Verkauf AS v
```

Listing 4.1: SQL: Dimensionalität

Unterschiedliche Aggregationshierarchien
Heterogenitäten aufgrund unterschiedlicher Aggregationshierachien in semantisch äquivalenten Dimensionen können mit SQL-Projektionen eliminiert werden. In Annahme von Star-Schemata, können jene Klassifikationsstufen, welche nicht in allen Dimensionen vorhanden sind, durch Projektion auf eine Dimensionstabelle ausgeblendet werden.

Beispiel 4.2: Die Klassifikationsstufe „Quartal“ der Dimension „Datum“ des MFUB-Schemas wird mit der SQL-Abfrage des Listing 4.2 aus der Dimensionstabelle ausgeblendet:

```
1 SELECT d.Datum, d.Monat, d.Jahr
2 FROM mfuB:Datum AS d
```

Listing 4.2: SQL: Unterschiedliche Aggregationshierarchien

Namenskonflikte
Heterogenitäten aufgrund unterschiedlicher Namen für Kenngrößen-, Klassifikationsstufen- oder (nicht-)dimensionale Attribute können in SQL mit dem Schlüsselwort `AS` im `SELECT` Abschnitt umbenannt werden.

Beispiel 4.3: Das Kenngrößen-Attribut „Dauer“ des MFUA-Schemas wird mit der SQL-Abfrage des Listing 4.3 in „Gesprächsdauer“ umbenannt:

```
1 SELECT ..., v.Dauer AS Gesprächsdauer, ...
2 FROM mfuA:Verkauf AS v
```

Listing 4.3: SQL: Namenskonflikt

Domänenkonflikte

Heterogenitäten aufgrund unterschiedlicher Domänen für Kenngrößen- oder nicht-dimensionale Attribute können mit SQL durch Einbinden einer Umrechnungsfunktion bereinigt werden.

Beispiel 4.4: Im MFUA-Schema werden Umsätze in Euro, im MFUB-Schema werden Umsätze in US-Dollar gespeichert. Die SQL-Abfrage des Listing 4.4 konvertiert die US-Dollar-Umsatzwerte des MFUB-Schemas in Euro:

```
1 SELECT ..., v.Umsatz / 1.3 AS Umsatz    // 1.3 = Umrechnungsfaktor
2 FROM mfuB:Verkauf AS v
```

Listing 4.4: SQL: Domänenkonflikt (Kenngrößen)

Ein Domänenkonflikt einer inneren Klassifikationsstufe einer Hierarchie kann, bei Annahme eines Star-Schemas, durch Ausblenden der Klassifikationsstufe mit einer SQL-Projektion auf die Dimensionstabelle eliminiert werden (vgl. Beispiel 4.2). Betrifft ein solcher Konflikt die Klassifikationsstufe niedrigster Granularität einer Hierarchie, so kann mit einer SQL-Abfrage ein neuer Würfel berechnet werden, indem eine Aggregation auf die nächst höhere Klassifikationsstufe erfolgt.

Beispiel 4.5: Um die Auflösung eines Domänenkonflikts einer Klassifikationsstufe niedrigster Granularität zu demonstrieren, wird angenommen, dass im MFUA-Schema in der Dimension „Datum“ auf der niedrigsten Granularität eine Klassifikationsstufe „Datum_std“ existiert. Die SQL-Abfrage des Listing 4.5 bereinigt diesen Konflikt, indem eine Aggregation auf „Datum“ erfolgt:

```
1 SELECT d.Datum,v.svnr,v.Mobilnetz,
2        SUM(v.Dauer),SUM(v.Umsatz_Telefonie),SUM(v.Umsatz_Sonstiges)
3 FROM mfuA:Verkauf AS v, mfuA:Datum AS d
4 WHERE v.Datum_std = d.Datum_std
5 GROUP BY d.Datum,v.svnr,v.Mobilnetz
```

Listing 4.5: SQL: Domänenkonflikt (unterste Klassifikationsstufe)

Schema-Instanz-Konflikte

Heterogenitäten aufgrund eines Schema-Instanz-Konflikts können mit SQL aufgrund der nicht-Unterstützung von Daten-Metadaten-Transformationen nur umständlich aufgelöst werden. Jedoch ist SchemaSQL sehr gut zur Restrukturierung von heterogenen Datenbank-Schemata geeignet, wo Daten eines Schemas, Metadaten eines anderen Schemas darstellen und umgekehrt. Daher können Schema-Instanz-Konflikte, deren Problematik in einem solchen Daten-Metadaten-Konflikt liegt, mit SchemaSQL-Abfragen elegant eliminiert werden.

Beispiel 4.6: Die SchemaSQL-Abfrage des Listing 4.6 transformiert das Schema der Fakttabelle „mfuB::Verkauf“ in das Schema der Fakttabelle „mfuA::Verkauf“:

```
1 CREATE VIEW bToA::Verkauf(Datum, svnr, Mobilnetz, Dauer, K) AS
2 SELECT V.Datum, V.Kunde, V.Mobilnetz, V.Gesprächsdauer, V.Umsatz
3 FROM mfuB::Verkauf V, V.Umsatzkategorie K
```

Listing 4.6: SchemaSQL: Schema-Instanz-Konfikt (MFUB → MFUA)

Alle Tupel der Fakttabelle „mfuB::Verkauf“, welche dieselben Werte für die Attribute {Datum, Kunde, Mobilnetz, Werbeaktion} haben, d.h. dieselben Koordinaten besitzen, werden in ein Output-Tupel zusammengeführt. Bei V handelt es sich um eine `tuple` Variable, welche über die Tupel der Fakttabelle iteriert, bei K um eine `domain` Variable, welche die Werte des Attributs „Umsatzkategorie“ der Fakttabelle enthält. Mit der `CREATE VIEW` Anweisung wird ein dynamisches Output-Schema erstellt, wobei die Variable K zwei Spalten produziert, nämlich „Umsatz_Telefonie“ und „Umsatz_Sonstiges“, welche den mit der entsprechenden Umsatzkategorie verbundenen Umsatz-Wert enthalten, z.B. `Umsatz_Telefonie=6,20`, `Umsatz_Sonstiges=3,50`. Anders ausgedrückt: das Schema `bToA::Verkauf` wird dynamisch als `bToA::Verkauf(Datum, svnr, Mobilnetz, Dauer,` K_1, ..., K_n`)` generiert, wobei K_1, ..., K_n die Werte des Attributs „Umsatzkategorie“ darstellt.

Beispiel 4.7: Die SchemaSQL-Abfrage des Listing 4.7 zeigt den umgekehrten Ansatz: das Schema der Fakttabelle „mfuA::Verkauf“ wird in das Schema der Fakttabelle „mfuB::Verkauf“ transformiert:

```
1 CREATE VIEW aToB::Verkauf(Datum, Kunde, Mobilnetz, Umsatzkategorie,
2                           Gesprächsdauer, Umsatz) AS
3 SELECT V.Datum, V.svnr, V.Mobilnetz, U, V.Dauer, V.U
4 FROM mfuA::Verkauf-> U, mfuA::Verkauf V
5 WHERE U <> 'Datum' AND U <> 'svnr' AND U <> 'Mobilnetz'
6       AND U <> 'Dauer'
```

Listing 4.7: SchemaSQL: Schema-Instanz-Konflikt (MFUA → MFUB)

Aus einem Tupel der Fakttabelle „mfuA::Verkauf“ werden zwei Output-Tupel erzeugt, d.h. je ein Output-Tupel für die Werte von „Umsatz_Telefonie“ und „Umsatz_Sonstiges“. Im `FROM` Abschnitt der Abfrage werden zwei Variablen deklariert: V ist eine `tuple` Variable, welche über die Tupel der Fakttabelle „mfuA::Verkauf“ iteriert; U ist eine `attr-name` Variable über die Attribute dieser Fakttabelle. Die Bedingungen im `WHERE` Abschnitt stellen sicher, dass U nur über die Umsatz-Attribute iteriert, also über „Umsatz_Telefonie“ und „Umsatz_Sonstiges“. Die Variable U im `SELECT` Abschnitt fügt die Namen der Umsatz-Kenngrößen-Attribute, also „Umsatz_Telefonie“ und „Umsatz_Sonstiges“ als Werte der Spalte „Umsatzkategorie“ ein, die Variable $V.U$ fügt den der jeweiligen Umsatzkategorie entsprechenden Umsatz-Wert ein. Letztendlich, entspricht jedes Output-Tupel `(V.Datum, V.svnr, V.Mobilnetz, U, V.Dauer, V.U)` dem Format der Fakttabelle „mfuB::Verkauf“.

Konflikte auf Instanzebene

Im Folgenden wird demonstriert, wie Konflikte auf Instanzebene mittels (Schema)SQL-Abfragen aufgelöst werden können.

Überlappende/Disjunkte Fakten

Bei überlappenden Fakten, d.h. Fakten mit identischen Primärschlüsseln (z.B. die Fakten mit dem Primärschlüssel {01.01.2006, 1234010180, MobCom}), können zwei Fälle aufgrund unterschiedlicher semantischer Beziehungen unterschieden werden: „Identitätsbeziehung“ und „Kontextbeziehung“ (vgl. Abschnitt 3.3.3). Fakten, welche sich aufgrund einer Identitätsbeziehung überlappen, können in SQL mit einem JOIN, wo nur die Fakten eines Schemas in das globale Schema übernommen werden, integriert werden. Aggregatsfunktionen können hier nicht verwendet werden.

Beispiel 4.8: Die SQL-Abfrage des Listing 4.8 behandelt überlappende Fakten aufgrund einer Identitätsbeziehung, indem nur jene aus dem MFUA-Schema in das globale Schema übernommen werden:

```
1 SELECT v1.Datum, v1.svnr, v1.Mobilnetz,
2        v1.Gesprächsdauer, v1.Umsatz_Telefonie, v1.Umsatz_Sonstiges
3 FROM mfuA:Verkauf AS v1, mfuB:Verkauf AS v2
4 WHERE v1.Datum = v2.Datum AND v1.svnr = v2.Kunde AND v1.Mobilnetz = v2.Mobilnetz
```

Listing 4.8: SQL: Überlappende Fakten (Identitätsbeziehung)

Fakten, welche in einer Kontextbeziehung zueinander stehen, können durch Anwendung einer Aggregatsfunktion zusammengeführt werden, falls die Schemata der Fakttabellen kompatibel sind.

Beispiel 4.9: Die SQL-Abfrage des Listing 4.9 behandelt überlappende Fakten mit Kontextbeziehung, indem die Werte der Kenngrößen der beiden Fakttabellen addiert werden:

```
1 SELECT v1.Datum, v1.svnr, v1.Mobilnetz,
2        SUM(v1.Gesprächsdauer + v2.Gesprächsdauer) AS Gesprächsdauer,
3        SUM(v1.Umsatz_Telefonie + v2.Umsatz_Telefonie) AS Umsatz_Telefonie,
4        SUM(v1.Umsatz_Sonstiges + v2.Umsatz_Sonstiges) AS Umsatz_Sonstiges
5 FROM mfuA:Verkauf AS v1, mfuB:Verkauf AS v2
6 WHERE v1.Datum = v2.Datum AND v1.svnr = v2.Kunde AND v1.Mobilnetz = v2.Mobilnetz
7 GROUP BY v1.Datum, v1.svnr, v1.Mobilnetz
```

Listing 4.9: SQL: Überlappende Fakten (Kontextbeziehung)

Für die Integration von disjunkten Fakten muss eine Entscheidung getroffen werden, wie bzw. ob diese im Ergebnis enthalten sein sollen. Bei gegebenen kompatiblen Schemata der Fakttabellen können die disjunkten Fakten mit den SQL-Mengenoperatoren UNION, JOIN, MINUS, FULL OUTER JOIN, etc. [KE04] integriert werden.

Beispiel 4.10: Die SQL-Abfrage des Listing 4.10 verwendet das Schlüsselwort UNION um jene Fakten zu vereinen, deren Primärschlüssel nicht in der jeweils anderen Fakttabelle enthalten ist, d.h. es werden nur die disjunkten Fakten selektiert:

```
1  SELECT v1.Datum, v1.svnr, v1.Mobilnetz,
2         v1.Gesprächsdauer, v1.Umsatz_Telefonie, v1.Umsatz_Sonstiges
3  FROM mfuA:Verkauf AS v1, mfuB:Verkauf AS v2
4  WHERE v1.Datum != v2.Datum OR v1.svnr != v2.Kunde OR v1.Mobilnetz != v2.Mobilnetz
5
6  UNION
7
8  SELECT v2.Datum, v2.Kunde, v2.Mobilnetz,
9         v2.Gesprächsdauer, v2.Umsatz_Telefonie, v2.Umsatz_Sonstiges
10 FROM mfuA:Verkauf AS v1, mfuB:Verkauf AS v2
11 WHERE v1.Datum != v2.Datum OR v1.svnr != v2.Kunde OR v1.Mobilnetz != v2.Mobilnetz
```

Listing 4.10: SQL: Disjunkte Fakten

Namenskonflikte

Für die Auflösung von Heterogenitäten aufgrund unterschiedlicher Namen für semantisch äquivalente Dimensionsinstanzen stehen in (Schema)SQL keine spezifischen Konstrukte zur Verfügung. Daraus folgt, dass die betroffenen Dimensionsinstanzen manuell umbenannt werden müssen.

Beispiel 4.11: Die Dimensionstabelle „Mobilnetz“ enthält eine Instanz „HandyTel“ im MFUA-Schema und eine Instanz „HandyTelInc“ im MFUB-Schema; beide bezeichnen jedoch dasselbe Mobilnetz. Im Listing 4.11 wird zunächst die Dimensionstabelle „Mobilnetz“ kopiert und anschließend die entsprechende Instanz der Kopie umbenannt:

```
1  SELECT * FROM mfuA:Mobilnetz INTO dupl:Mobilnetz
2
3  UPDATE dupl:Verkauf SET Mobilnetz = 'HandyTelInc' WHERE Mobilnetz = 'HandyTel'
```

Listing 4.11: SQL: Namenskonflikte (Instanzen)

Überlappende/Disjunkte Dimensionsinstanzen

Für überlappende/disjunkte Dimensionsinstanzen gilt dasselbe wie für Fakten, d.h. eine Integration mit den genannten SQL-Mengenoperatoren ist möglich. Da überlappende Dimensionsinstanzen in einer Identitätsbeziehung zueinander stehen, können keine Aggregatsfunktionen angewendet werden. Für die Formulierung von SQL-Abfragen wird auf die Beispiele 4.8 und 4.10 verwiesen.

Heterogene roll-up Beziehungen

Für die Auflösung von Konflikten aufgrund heterogener roll-up Beziehungen stehen keine spezifischen Konstrukte in SchemaSQL zur Verfügung. Daraus folgt, dass die betroffenen Dimensionsinstanzen manuell umbenannt werden müssen. Für die Formulierung von SQL-Abfragen wird auf das Beispiel 4.11 verwiesen.

4.3.1.3 Exkurs: Horizontale Aggregation

Da die horizontale Aggregation eine mächtige SQL-Erweiterung darstellt, soll diese durch folgendes Beispiel veranschaulicht werden:

Beispiel 4.12: Die SchemaSQL-Abfrage des Listing 4.12 berechnet den Gesamtumsatz pro Tupel der Fakttabelle „mfuA::Verkauf“, also die Summe von „Umsatz_Telefonie“ und „Umsatz_Sonstiges“:

```
1 SELECT V.Datum,V.svnr,V.Mobilnetz,V.Dauer,SUM(V.U)
2 FROM mfuA::Verkauf-> U,mfuA::Verkauf V
3 WHERE U <> 'Datum' AND U <> 'svnr' AND U <> 'Mobilnetz'
4       AND U <> 'Dauer'
5 GROUP BY V.Datum, V.svnr, V.Mobilnetz, V.Dauer
```

Listing 4.12: SchemaSQL - Exkurs: Horizontale Aggregation

Diese Abfrage ähnelt der des Listing 4.7: erneut werden eine `tuple` und eine `attr-name` Variable verwendet; die Bedingungen im `WHERE` Abschnitt stellen sicher, dass die Variable *U* nur über Umsatz-Attribute iteriert.

Mit Hilfe der horizontalen Aggregation können somit Aggregationen für eine beliebige, unbestimmte Menge von Attributen berechnet werden.

4.3.1.4 Fazit

Wie aus der Analyse der Anwendbarkeit von SchemaSQL für die Integration von DWS hervorgeht, eignet sich diese Abfragesprache nur bedingt für diese Aufgabe, da die Integration von heterogenen Würfeln eine große Anzahl von (Schema)SQL-Abfragen erfordern kann. Die Komplexität der Kombination der Ergebnisse der Teil-Abfragen ist als hoch einzuschätzen. Darüber hinaus kann SchemaSQL nur für die Integration von ROLAP-Systemen (vgl. Abschnitt 2.4.2) eingesetzt werden, da kein konzeptuelles, multidimensionales Datenmodell unterstützt wird. Hingegen können Schema-Instanz-Konflikte aufgrund der Fähigkeit, Daten zu Metadaten und umgekehrt zu transformieren, sehr elegant bereinigt werden. Obwohl sich SchemaSQL bezüglich der Aggregationsfunktionalität als mächtig erweist (z.B. horizontale Aggregation), erschwert die nicht-Unterstützung eines multidimensionalen Datenmodells einen Einsatz für die Integration von DWS.

4.3.2 nD-SQL

nD-SQL [GL98] ist eine SQL-kompatible Abfragesprache, welche komplexe Schema-Restrukturierungen in einem MDBS unterstützt. Hinsichtlich der Fähigkeit zu Daten-Metadaten-Transformationen und der Syntax weist nD-SQL Gemeinsamkeiten mit SchemaSQL (vgl. Abschnitt 4.3.1) auf. Darüber hinaus bietet nD-SQL Unterstützung für komplexe OLAP-Abfragen.

4.3.2.1 Theoretische Konzepte

Die Beschreibung der nD-SQL zugrunde liegenden theoretischen Konzepte gliedert sich in zwei Teile: zunächst werden die für Abfragen in einem MDBS erforderlichen Konzepte und Syntaxelemente erläutert, anschließend jene für OLAP-Abfragen.

Konzepte und Syntaxelemente für MDBS-Abfragen

Analog zu SchemaSQL, erlaubt nD-SQL Variablen, welche über Datenbanknamen, über Relationennamen einer Datenbank, über Tupel einer Relation sowie über Attributnamen einer Relation iterieren. Die Syntax und Semantik der Variablen-Deklarationen entspricht dabei weitestgehend jener von SchemaSQL, weshalb auf deren Erläuterung verzichtet wird und im Folgenden die Bezeichnungen aus SchemaSQL verwendet werden.

Das nD-SQL zugrunde liegende Datenmodell kennt komplexe Attribute, die aus mehreren Attributen bestehen, welchen dieselben Metadaten zugeordnet sind. Diesen, sowie Relationen, liegen so genannte „Kriterien“ und „Konzepte“ zugrunde, welche die ihnen zuzuordnenden Metadaten repräsentieren. In der Fakttabelle „mfuA::Verkauf“ des Rahmenbeispiels können die beiden Attribute „Umsatz_Telefonie“ und „Umsatz_Sonstiges“ als komplexes Attribut angesehen werden, welches dem Kriterium „Umsatzkategorie“ (mit den Werten „Umsatz_Telefonie“ und „Umsatz_Sonstiges“) und dem Konzept „Umsatz“ (mit den Werten der einzelnen Umsätze) zuzuordnen ist. Für detailliertere Informationen über das nD-SQL-Modell, wird auf [GL98] verwiesen.

nD-SQL erweitert einige Abschnitte der SQL-Syntax um zusätzliche semantische Eigenschaften und Konstrukte:

`FROM` nD-SQL erlaubt zusätzlich die Deklaration von Variablen über Datenbanknamen, Relationennamen und Attributnamen.

`WHERE` Hier sind zusätzliche Schlüsselwörter verfügbar um sicherzustellen, dass `rel-name` Variablen und `attr-name` Variablen über eine homogene Menge von Schemaobjekten iterieren, d.h. über Relationen bzw. Attribute mit demselben zugeordneten Kriterium (Schlüsselwort `HASA`) bzw. Konzept (Schlüsselwort `ISA`).

`SELECT` Um komplexe Attribute und Relationen zu erzeugen, können Daten in Kriterien, d.h. in Metadaten, mit Hilfe des Schlüsselworts `FOR` transformiert werden. Das Schlüsselwort `FOR` ersetzt die `CREATE VIEW` Anweisung aus SchemaSQL und dient der Erzeugung dynamischer Output-Schemata.

Konzepte und Syntaxelemente für OLAP-Abfragen

Neben den für Abfragen in einem MDBS erforderlichen Schema-Restrukturierungen, bietet nD-SQL auch erweiterte OLAP-Funktionalitäten, insbesondere Aggregationen für beliebige GROUP-BYs. Benutzer sind oft an spezifischen GROUP-BY-Kombinationen interessiert, welche nicht vom ROLLUP-Operator abgedeckt werden. Andererseits berechnet der CUBE-Operator [GCB+97] Aggregationen, welche der Benutzer nicht benötigt. Aus diesem Grund erlaubt nD-SQL die Spezifikation von beliebigen GROUP-BY-Kombinationen für beliebige Granularitäten.

Um dies zu ermöglichen, wurde die nD-SQL-Syntax um eine `DIM` Variable im `FROM` Abschnitt erweitert, welche über die Dimensionen eines Würfels iteriert. Für diese Variablen können zusätzlich Bedingungen mit den Vergleichsoperatoren $=, \leq, <, >, \geq, \neq$ definiert werden. Weiters existiert die spezielle Konstante `NONE`, welche die Klassifikationsstufe „all“ repräsentiert. Mit dem Schlüsselwort `IN` können zulässige Wertebereiche für die `DIM` Variablen, d.h. zulässige Dimensionen, festgelegt werden.

Anwendungsbeispiele für die erläuterten Konzepte und Syntaxelemente werden im folgenden Abschnitt sowie im Exkurs des Abschnitts 4.3.2.3 präsentiert und erläutert.

4.3.2.2 Anwendung für die Integration von DWS

Im Zuge der Analyse von nD-SQL hinsichtlich der Eignung für die Integration von DWS stellte sich heraus, dass diese Abfragesprache ausreichend mächtig ist um Schema-Instanz-Konflikte aufzulösen. Die übrigen der im Abschnitt 3.3.3 beschriebenen Heterogenitäten können zum Teil nur umständlich durch eine Menge von SQL-Abfragen bereinigt werden. Im Folgenden wird daher anhand des Rahmenbeispiels der Abbildung 3.4 sowie der Tabellen 3.2 und 3.3 gezeigt, wie Schema-Instanz-Konflikte mit nD-SQL-Abfragen aufgelöst werden können. Bezüglich der Auflösung der übrigen Heterogenitäten mit SQL-Abfragen wird auf den Abschnitt 4.3.1.2 zu SchemaSQL verwiesen.

Schema-Instanz-Konflikte
Aufgrund der Fähigkeit Daten und Metadaten simultan zu verarbeiten, eignet sich nD-SQL für die Auflösung von Schema-Instanz-Konflikten.

Beispiel 4.13: Die nD-SQL-Abfrage des Listing 4.13 transformiert das Schema der Fakttabelle „mfuB::Verkauf“ in das Schema der Fakttabelle „mfuA::Verkauf“:

```
1 SELECT V.Datum,V.Kunde,V.Mobilnetz,V.Gesprächsdauer,
2        V.Umsatz AS V.Umsatzkategorie FOR V.Umsatzkategorie
3 FROM mfuB::Verkauf V
```

Listing 4.13: nD-SQL: Schema-Instanz-Konflikt (MFUB → MFUA)

Im Unterschied zur äquivalenten SchemaSQL-Abfrage (vgl. Listing 4.6) muss hier nur eine Variable, nämlich eine `tuple` Variable, deklariert werden. Die Werte des Attributs „Umsatzkategorie“, nämlich „Umsatz_Telefonie“ und „Umsatz_Sonstiges“, werden mit Hilfe des Konstrukts `V.Umsatz AS V.Umsatzkategorie FOR V.Umsatzkategorie` zu separaten Umsatz-Kenngrößen-Attributen, d.h. zu Kriterien. Im Gegensatz zum SchemaSQL-Äquivalent ist hier keine Definition einer Sicht erforderlich.

Beispiel 4.14: Die nD-SQL-Abfrage des Listing 4.14 zeigt den umgekehrten Ansatz: das Schema der Fakttabelle „mfuA::Verkauf“ wird in das Schema der Fakttabelle „mfuB::Verkauf“ transformiert:

```
1 SELECT V.Datum,V.svnr,V.Mobilnetz,V.Dauer,
2        K.Umsatzkategorie, V.K AS Umsatz
3 FROM mfuA::Verkauf V, mfuA::Verkauf -> K
4 WHERE K ISA Umsatz
```

Listing 4.14: nD-SQL: Schema-Instanz-Konflikt (MFUA → MFUB)

Zusätzlich zur `tuple` Variable muss bei dieser Abfrage eine `attr-name` Variable für die Fakttabelle „mfuA::Verkauf“ deklariert werden. Im `WHERE` Abschnitt wird eine Bedingung definiert, welche sicherstellt, dass K nur Attribute enthält, welche dem Konzept „Umsatz“ zuzuordnen sind, d.h. die Umsatz-Kenngrößen-Attribute „Umsatz_Telefonie“ und „Umsatz_Sonstiges“ des Kriteriums `K.Umsatzkategorie`. Die Werte des Kriteriums `K.Umsatzkategorie` werden in eine eigene Spalte extrahiert. Das Konstrukt `V.K AS Umsatz` bewirkt, dass die Umsatzwerte der Attribute „Umsatz_Telefonie“ und „Umsatz_Sonstiges“ entsprechend dem Wert in `K.Umsatzkategorie` in ein Kenngrößen-Attribut „Umsatz“ geschrieben werden.

Wie aus den beiden Beispielen ersichtlich, weist nD-SQL hinsichtlich der Auflösung von Schema-Instanz-Konflikten Gemeinsamkeiten mit SchemaSQL auf.

4.3.2.3 Exkurs: Beliebige GROUP-BY Kombinationen

nD-SQL erweist sich bezüglich der Aggregationsfähigkeiten, insbesondere aufgrund der Unterstützung beliebiger GROUP-BY-Kombinationen, als mächtig. Dies soll folgendes Beispiel verdeutlichen:

Beispiel 4.15: Die nD-SQL-Abfrage des Listing 4.15 berechnet die Summe der Telefonie-Umsätze pro Monat, Tarif und Mobilnetz, pro Monat und Tarif und pro Monat für MFUA:

```
1 SELECT X,Y,Z, SUM(V.Umsatz_Telefonie)
2 FROM mfuA::Verkauf V,mfuA::Datum D,mfuA::Kunde K, DIM X, Y, Z
3 WHERE V.Datum = D.Datum AND V.svnr = K.svnr
4       AND DIM X IN {D.Monat}, DIM Y IN {K.Tarif,NONE}
5       AND DIM Z IN {K.Mobilnetz,NONE} AND X < Y < Z
6 GROUP BY X,Y,Z
```

Listing 4.15: nD-SQL - Exkurs: Beliebige GROUP-BYs

Im `FROM` Abschnitt werden drei `DIM` Variablen deklariert, für welche im `WHERE` Abschnitt Bedingungen formuliert werden: die Variable X kann nur die Klassifikationsstufe „Monat“ der Dimension „Datum“ annehmen, die Variable Y die Klassifikationsstufe „Tarif“ der Dimension „Kunde“ sowie die spezielle Konstante `NONE` (= Klassifikationsstufe „all“) und die Variable Z die Klassifikationsstufe „Mobilnetz“ der Dimension „Mobilnetz“ sowie die spezielle Konstante `NONE` (= Klassifikationsstufe „all“). Die Bedingung `X < Y < Z` bewirkt die Berechnung der Summe für die genannten GROUP-BY-Kombinationen. Der Benutzer kann also Bedingungen für beliebige GROUP-BY-Kombinationen formulieren um beliebige Zwischenresultate zu erhalten.

4.3.2.4 Fazit

Die im Zuge der Integration von DWS möglicherweise auftretenden Heterogenitäten können mit nD-SQL nur durch eine Kombination einzelner Abfragen überwunden werden. nD-SQL ermöglicht durch die Ausrichtung auf Daten-Metadaten-Transformationen eine elegante Auflösung von Schema-Instanz-Konflikten. Die erweiterten Aggregationsfähigkeiten sowie die in [GL98] verwendete Charakterisierung als „multidimensionale Abfragesprache“ können nicht darüber hinwegtäuschen, dass nD-SQL kein multidimensionales Datenmodell zugrunde liegt; nD-SQL basiert, wie SchemaSQL, auf dem relationalen Datenmodell. Aus diesem Grund kann nD-SQL nur für die Integration von ROLAP-Systemen (vgl. Abschnitt 2.4.2) verwenden werden. Dies ist als das Hauptdefizit von nD-SQL und SchemaSQL anzusehen, welches einen Einsatz zur Integration von DWS erschwert.

4.3.3 CQL

CQL (Cube-Query-Language) [BL97] ist eine Abfragesprache für Data Warehouses. Im Gegensatz zu SchemaSQL (vgl. Abschnitt 4.3.1) und nD-SQL (vgl. Abschnitt 4.3.2) besteht der Zweck dieser Abfragesprache nicht in der Integration von DBS bzw. DWS. Da CQL jedoch speziell für OLAP-Anwendungen entwickelt wurde und ein multidimensionales Datenmodell (CROSS-DB) [LRT96] implementiert, werden in diesem Abschnitt die grundlegenden Konzepte dieser Abfragesprache erläutert und ihre Eignung für die Integration von DWS beurteilt.

4.3.3.1 Theoretische Konzepte

Eine innovative Eigenschaft von CQL ist die Ausrichtung der Sprache auf die Unterstützung von Client-Server-Anwendungen. Am Client formuliert ein Benutzer eine Abfrage, welche an einen Server gesendet wird. Am Server wird das Ergebnis berechnet und eine Referenz auf dieses an den Client retourniert (CQL - Berechnung). Mit Hilfe der `SHOW` Anweisung kann der Benutzer bereits berechnete Abfrageergebnisse aktivieren und die entsprechenden Daten zum Client transferieren, wo diese gemäß der in der `SHOW` Anweisung definierten Präsentationseigenschaften angezeigt werden (CQL - Präsentation). Somit liegt beim Server die Aufgabe der Berechnung des Abfrageergebnisses und beim Client die Aufgabe der Ergebnispräsentation.

Im Folgenden werden die wichtigsten Eigenschaften des CQL zugrunde liegenden multidimensionalen Datenmodells erläutert. Im Anschluss werden die Konzepte und Syntaxelemente für die Ergebnisberechnung und -präsentation behandelt.

CROSS-DB Datenmodell

Das CROSS-DB(Classification-oriented, Redundancy-based Optimization of Statistical and Scientific DataBases)-Datenmodell [LRT96] kennt die Konstrukte des im Abschnitt 2.3 vorgestellten, multidimensionalen Datenmodells, wodurch CROSS-DB die Handhabung von Fakten und Dimensionen samt Hierarchien ermöglicht. Darüber hinaus erlaubt es die Spezifikation von Eigenschaftswerten (= nicht-dimensionale Attribute), so genannten „features“, für einzelne Klassifikationsstufen, wodurch diese genauer beschrieben werden können. In diesem Zusammenhang ist zu erwähnen, dass die „features“ top-down entsprechend der Dimensionshierarchie, d.h. von der Klassifikationsstufe gröbster Granularität bis zur Klassifikationsstufe feinster Granularität, vererbt werden.

CQL - Berechnung

Die CQL-Syntax zur Formulierung einer Abfrage orientiert sich an der SQL-Notation, wobei gegenüber SQL zusätzliche bzw. anders lautende Konstrukte existieren. Die Semantik mancher

syntaktisch äquivalenter Konstrukte unterscheidet sich jedoch aufgrund der Multidimensionalität von CQL von den entsprechenden SQL-Konstrukten. Im Folgenden wird die CQL-Syntax und -Semantik näher erläutert, insbesondere wird gezeigt, wie der Kontext einer Abfrage spezifiziert wird, wie Aggregatsfunktionen angewendet, wie geschachtelte Sub-Abfragen definiert und wie horizontale Analysen durchgeführt werden können.

Abfrage-Kontext
Folgende CQL-Konstrukte dienen der Spezifikation des Abfrage-Kontexts, welcher den zu berechnenden Würfel beschreibt:

`SELECT` Im Gegensatz zu SQL, erfordert dieser Abschnitt keine Anführung der dimensionalen Attribute nach denen eine Gruppierung erfolgt, d.h. es werden nur die Kenngrößen-Attribute angeführt.

`FROM` Hier werden die Dimensionen des Würfels spezifiziert um eine bestimmte Kenngröße zu identifizieren. Alle für die Identifizierung einer Kenngröße erforderlichen Dimensionen müssen angegeben werden (entspricht dem Primärschlüssel einer Fakttabelle). Dadurch, dass CQL unabhängig von der Implementierung des multidimensionalen Datenmodells ist, entfällt die Angabe der Fakt- und Dimensionstabellen.

`WHERE` Hier werden OLAP-Operationen wie slice (Selektion) oder dice (Projektion) spezifiziert. Selektionen und Projektionen erfolgen durch die Definition von Bedingungen für die Dimensionen, wobei jene Dimensionen, für welche keine Bedingungen definiert werden, ausgeblendet werden (= Projektion bzw. roll-up auf Klassfikationsstufe „all“). Um die Selektionen zu verfeinern, erlaubt CQL die Definition von Bedingungen für „features“.

`RESTRICT` Dieses Konstrukt entspricht dem `HAVING` Konstrukt aus SQL, d.h. es wird der zulässige Wertebereich für die quantitativen Kenngrößen-Attribute spezifiziert.

Würfel-Operationen
Das CQL zugrunde liegende CROSS-DB-Datenmodell unterscheidet zellenorientierte und aggregationsorientierte Operatoren für multidimensionale Würfel. Mit den zellenorientierten Operatoren, z.B. für arithmetische Operationen, können einzelne Kenngrößen eines ausgewählten Würfel-Ausschnitts zusammengeführt werden (z.B. Preis * Menge = Umsatz). Aggregationsorientierte Operatoren, wie SUM, AVG, COUNT, MIN, MAX, verdichten einen Input-Würfel auf eine gröbere Granularität. Die Spezifikation der Granularität erfolgt mit dem `UPTO` Konstrukt, welches mit dem `GROUP BY` Konstrukt aus SQL vergleichbar ist.

Sub-Abfragen
CQL erlaubt die Spezifikation von Sub-Abfragen mit dem `WITH` Konstrukt um Sub-Würfel darzustellen. Die Variablen (Kenngrößen-Attribute) der Sub-Abfragen können dabei in der äußeren Abfrage verwendet werden. Die Sub-Abfragen erben den `FROM` und `WHERE` Abschnitt von der äußeren Abfrage. Wird jedoch in den Sub-Abfragen ein `FROM` Abschnitt spezifiziert, müssen die enthaltenen Dimensionen eine Teilmenge der im äußeren `FROM` Abschnitt spezifizierten Dimensionen sein.

Horizontale Analysen
Horizontale Analysen können in CQL mit dem so genannten „feature split“ (`BY` Konstrukt) durchgeführt werden, wodurch aggregierte Werte detaillierter aufgeschlüsselt werden können.

CQL - Präsentation

CQL bietet umfangreiche Möglichkeiten für die Darstellung der Ergebnistabellen, welche mit den in der Tabular Algebra (vgl. Abschnitt 4.2.7) vorgesehenen vergleichbar sind [GLS96]. Die Präsentation der Abfrageergebnisse wird mit einer `SHOW` Anweisung, welche folgende Konstrukte umfasst, angestoßen:

`FROM` Hier werden die Dimensionen des zu präsentierenden Würfels angeführt, wobei diese den in der Abfrage spezifizierten entsprechen müssen. Dieser Abschnitt dient der Definition von Namen, welche in den weiteren Konstrukten verwendet werden können.

`STRUCTURE` Hier wird die Struktur der Ergebnistabelle spezifiziert. Aufgrund der Zweidimensionalität einer Tabelle, gliedert sich `STRUCTURE` in zwei Sub-Abschnitte:

- `HEADER` Hier wird das horizonale Erscheinungsbild einer Tabelle spezifiziert, d.h. die Spalten-Namen, welche auch geschachtelt sein können. Mit den Schlüsselwörtern `TOTALS` und `CUM` können Spalten-Zwischensummen bzw. kumulierte Summen in das Ergebnis integriert werden.
- `STUB` Hier wird das vertikale Erscheinungsbild einer Tabelle spezifiziert, d.h. die Zeilen-Namen, welche auch geschachtelt sein können. Mit den Schlüsselwörtern `TOTALS` und `CUM` können Zeilen-Zwischensummen bzw. kumulierte Summen in das Ergebnis integriert werden.

`SORTED BY` Hier kann das Ergebnis nach beliebigen Spalten und Zeilen sortiert werden.

Im folgenden Abschnitt wird CQL hinsichtlich der Eignung für die Integration von DWS analysiert. Eine beispielhafte Anwendung der beschriebenen Besonderheiten von CQL folgt im Exkurs des Abschnitts 4.3.3.3.

4.3.3.2 Anwendung für die Integration von DWS

CQL zielt nicht auf die Integration von DBS bzw. DWS ab, sondern implementiert ein multidimensionales Datenmodell um Abfragen auf Würfel zu ermöglichen. Trotz der Ähnlichkeit hinsichtlich der Syntax ist CQL nicht SQL-kompatibel, wodurch die SQL-Fähigkeiten der zu integrierenden Komponenten-DWS nicht genutzt werden können.

Soweit aus [BL97] zu ermitteln war, können mit CQL nur zwei der im Abschnitt 3.3.3 beschriebenen Heterogenitäten aufgelöst werden, nämlich „Dimensionalität“ und „Domänenkonflikte“. Im Folgenden wird demonstriert, wie diese Heterogenitäten mit CQL-Abfragen eliminiert werden können. Für die übrigen Heterogenitäten wird kurz erläutert, warum diese mit CQL nicht handhabbar sind.

Dimensionalität
Heterogenitäten aufgrund unterschiedlicher Dimensionalität der Würfel können mit CQL-Abfragen aufgelöst werden, indem für die auszublendende Dimension keine Bedingung im `WHERE` Abschnitt der Abfrage formuliert wird.

Beispiel 4.16: Die im MFUB-Schema zusätzlich vorhandene Dimension „Werbeaktion“ wird mit der CQL-Abfrage des Listing 4.16 ausgeblendet:

```
1 SELECT Gesprächsdauer, Umsatz
2 FROM Datum D, Kunde K, Mobilnetz M, Umsatzkategorie U, Werbeaktion W
3 WHERE D.Jahr = '2006' OR ... AND K.Tarif = '2For0' OR ...
4       AND M.Mobilnetz = 'MobCom' OR ...
```

Listing 4.16: CQL: Dimensionalität

Domänenkonflikte (Kenngrößen)
Heterogenitäten aufgrund von Domänenkonflikten bei Kenngrößen-Attributen können durch Einbinden einer arithmetischen Operation im `SELECT` Abschnitt einer Abfrage bereinigt werden.

Beispiel 4.17: Im MFUA-Schema werden Umsätze in Euro, im MFUB-Schema werden Umsätze in US-Dollar gespeichert. Die CQL-Abfrage des Listing 4.17 konvertiert die US-Dollar-Umsatzwerte des MFUB-Schemas in Euro:

```
1 SELECT Gesprächsdauer, Umsatz / 1.3    // 1.3 = Umrechnungsfaktor
2 FROM Datum D, Kunde K, Mobilnetz M, Umsatzkategorie U, Werbeaktion W
3 WHERE ...
```

Listing 4.17: CQL: Domänenkonflikt (Kenngrößen)

Betrifft ein Domänenkonflikt die Klassifikationsstufe niedrigster Granularität einer Hierarchie, so kann mit einer CQL-Abfrage ein neuer Würfel berechnet werden, indem eine Aggregation auf die nächst höhere Klassifikationsstufe erfolgt.

Beispiel 4.18: Um die Auflösung eines Domänenkonflikts einer Klassifikationsstufe niedrigster Granularität zu demonstrieren, wird angenommen, dass im MFUA-Schema in der Dimension „Datum“ auf der niedrigsten Granularität eine Klassifikationsstufe „Datum_std“ existiert. Die CQL-Abfrage des Listing 4.18 bereinigt diesen Konflikt, indem eine Aggregation auf „Datum“ erfolgt:

```
1 SELECT SUM(Dauer), SUM(Umsatz_Telefonie), SUM(Umsatz_Sonstiges)
2 FROM Datum D, Kunde K, Mobilnetz M
3 WHERE ...
4 UPTO D.Datum, K.svnr, M.Mobilnetz
```

Listing 4.18: CQL: Domänenkonflikt (unterste Klassifikationsstufe)

Mit Hilfe des `UPTO` Konstrukts wird der Würfel in der Dimension „Datum“ auf die Klassifikationsstufe „Datum“ aggregiert.

Übrige Heterogenitäten
Mit CQL sind nur Abfragen auf vollständige Würfel möglich. Somit können keine Abfragen auf Dimensionen formuliert werden, wodurch Heterogenitäten im Dimensions-Kontext (vgl.

Tabelle 3.1) nicht auflösbar sind. Da CQL-Abfragen auch nur auf einen einzelnen Würfel abgesetzt werden können, d.h. es ist kein „drill-across“ möglich, können überlappende/disjunkte Fakten nicht gehandhabt werden. Darüber hinaus besteht, soweit bekannt, keine Möglichkeit um Attribute umzubenennen. Aufgrund der Tatsache, dass CQL nicht für Integrationsaufgaben konzipiert wurde, werden Daten-Metadaten-Transformationen nicht unterstützt, wodurch keine Schema-Instanz-Konflikte eliminiert werden können.

Im folgenden Exkurs werden die besonderen Fähigkeiten von CQL anhand von Abfragen präsentiert, welche sich auf das konzeptuelle Modell des Rahmenbeispiels (vgl. Abbildung 3.4) beziehen.

4.3.3.3 Exkurs

In diesem Abschnitt wird gezeigt, wie Sub-Abfragen zur Repräsentation von Sub-Würfeln formuliert werden, wie Detailanalysen mit Hilfe des „feature split“ funktionieren und wie die Präsentation von Abfrageergebnissen gesteuert werden kann.

Sub-Abfragen
In einer CQL-Abfrage können mit dem `WITH` Konstrukt Sub-Abfragen formuliert werden, welche Sub-Würfel repräsentieren.

Beispiel 4.19: Die Abfrage des Listing 4.19 errechnet das Verhältnis der Kenngrößen „Umsatz_Telefonie“ zu „Umsatz_Sonstiges“ pro Monat, Tarif und Mobilnetz (MFUA-Schema) mit Hilfe von Sub-Abfragen:

```
1 SELECT UTEL / USONST
2 FROM Datum D, Kunde K, Mobilnetz M
3 WITH (SELECT SUM(Umsatz\_Telefonie) AS UTEL
4       UPTO D.Monat, K.Tarif, M.Mobilnetz),
5      (SELECT SUM(Umsatz\_Sonstiges) AS USONST
6       UPTO D.Monat, K.Tarif, M.Mobilnetz)
```

Listing 4.19: CQL - Exkurs: Sub-Abfragen

Im `FROM` Abschnitt werden die Dimensionen {Datum, Kunde, Mobilnetz} des Würfels spezifiziert. Im `WITH` Abschnitt werden zwei Sub-Würfel deklariert, welche die Summen der Telefonie- und der sonstigen Umsätze enthalten. Diese Werte werden in der äußeren Abfrage referenziert und für die Berechnung des Endergebnisses, des Verhältnisses „Umsatz_Telefonie“ zu „Umsatz_Sonstiges“, verwendet.

Feature split
In einer CQL-Abfrage kann der so genannte „feature split“ genutzt werden, um horizontale Analysen durchzuführen. Dabei können aggregierte Werte anhand nicht-dimensionaler Attribute (features) detaillierter aufgeschlüsselt werden.

Beispiel 4.20: Die Abfrage des Listing 4.20 verwendet einen „feature split“: es wird die Summe der Umsätze pro Monat (des Jahres 2006) und Tarif (für die Tarife '2For0' und 'FullCom') berechnet, wobei die Umsätze entsprechend dem nicht-dimensionalen Attribut „Altersgruppe“ aufgeschlüsselt werden (Schema MFUB):

```
1 SELECT SUM(Umsatz)
2 FROM Datum D, Kunde K, Mobilnetz M
3 WHERE D.Jahr = '2006' AND K.Tarif = '2For0' OR K.Tarif = 'FullCom'
4 UPTO D.Month, K.Tarif
5 BY K->Altersgruppe
```

Listing 4.20: CQL - Exkurs: Feature Split

Die Dimensionen werden im `FROM` Abschnitt spezifiziert; die Gruppierungen im `UPTO` Abschnitt. Im `WHERE` Abschnitt werden slice-Operationen (`D.Jahr = '2006' AND K.Tarif = '2For0' OR K.Tarif = 'FullCom'`) sowie eine dice-Operation (Dimension „Mobilnetz“ wird ausgeblendet, da keine entsprechende Bedingung formuliert wurde) durchgeführt. Im `BY` Abschnitt wird festgelegt, dass die Umsätze nach Altersgruppen aufgeschlüsselt werden sollen.

Ergebnispräsentation

Die am Client auszuführende Präsentation der Abfrageergebnisse wird mit der `SHOW` Anweisung angestoßen, wobei mit unterschiedlichen Konstrukten die Art der Präsentation gesteuert werden kann.

Beispiel 4.21: Die CQL-Abfrage des Listing 4.21 bereitet die Präsentation der Abfrage des Listing 4.17 auf: Als Spaltenbeschriftungen werden die einzelnen Monate, als Zeilenbeschriftungen die beiden Tarife und geschachtelt die ihnen zugeordneten Altersgruppen samt Zwischensummen verwendet:

```
1 SHOW
2 FROM Datum D, Kunde K, Mobilnetz M
3 STRUCTURE
4   HEADER D.Month
5   STUB K.Tarif (K.Altersgruppe TOTALS)
```

Listing 4.21: CQL - Exkurs: Ergebnispräsentation

Die Ergebnistabelle aufgrund der obigen Abfrage hat folgende Struktur:

Verkauf		**01/06**	**03/06**	**04/06**	**04/06**	**...**
2For0	**≤10**					
	11-20					
	21-30					
	...					
	$\sum$					
MaxKlax	**≤10**					
	11-20					
	21-30					
	...					
	$\sum$					

Tabelle 4.13: CQL: Ergebnispräsentation

4.3.3.4 Fazit

Wie aus den Erläuterungen und den Beispiel-Abfragen ersichtlich, bietet CQL durch die Implementierung eines multidimensionalen Datenmodells umfangreiche Unterstützung für OLAP-Anwendungen. Jedoch ist CQL nicht für die Integration von DBS bzw. DWS konzipiert worden, weshalb der Großteil der im Abschnitt 3.3.3 beschriebenen Heterogenitäten nicht eliminiert werden kann. Aufgrund der nicht-Unterstützung von Daten-Metadaten-Transformationen ist z.B. die Auflösung von Schema-Instanz-Konflikten nicht durchführbar. Nichtsdestotrotz stellt CQL eine mächtige multidimensionale Abfragesprache dar, deren Ideen und Konzepte (z.B. Sub-Abfragen für Sub-Würfel) für das Design einer Abfragesprache für die Integration von DWS verwendet werden können.

4.3.4 Beurteilung

Wie aus der Analyse der drei für die Integration von DWS als am geeignetsten erscheinenden Abfragesprachen SchemaSQL (vgl. Abschnitt 4.3.1), nD-SQL (vgl. Abschnitt 4.3.2) und CQL (vgl. Abschnitt 4.3.3) hervorgeht, ist keine im Stande sämtliche im Abschnitt 3.3.3 diskutierten Heterogenitäten in einer einzelnen Abfrage aufzulösen um einen globalen, instanziierten Würfel zu erzeugen. Vielmehr ist es erforderlich, eine große Anzahl von Teil-Abfragen zu kombinieren um ein integriertes Ergebnis zu erhalten.

SchemaSQL und nD-SQL basieren trotz der gegenüber SQL erweiterten Aggregationsfähigkeiten auf dem relationalen Datenmodell. Aus diesem Grund müssen für die Auflösung der Heterogenitäten zwischen Fakten und Dimensionen samt Hierarchien, umständliche workarounds, z.B. mit SQL-Abfragen, entwickelt werden. Darüber hinaus besteht bei diesen Sprachen die Problematik, dass sie nur für ROLAP-Implementierungen verwendet werden können. Die Stärken der beiden Abfragesprachen liegen in der Unterstützung von Schema-Restrukturierungen in einem heterogenen MDBS, indem Daten zu Metadaten und umgekehrt transformiert werden.

Demgegenüber basiert CQL auf einem multidimensionalen Datenmodell (CROSS-DB), wodurch von der DW-Implementierung unabhängige Abfragen formuliert werden können. CQL unterstützt detailierte OLAP-Analysen, u.a. durch vielfältige Darstellungsmöglichkeiten der Analyseergebnisse. Die Problematik bei der Verwendung von CQL für die Integration von DWS resultiert daraus, dass diese Abfragesprache nicht für Integrationsaufgaben konzipiert wurde und somit z.B. keine Daten-Metadaten-Transformationen erlaubt.

Um die Komplexität zu reduzieren, welche aus der Kombination vieler Teil-Abfragen zur Berechnung eines integrierten Ergebnisses resultiert, wäre es wünschenswert, eine Abfragesprache nutzen zu können, mit welcher mit nur einer Abfrage ein integriertes Ergebnis berechnet werden kann. Nach bestem Wissen des Autors ist die im Kapitel 5 vorgestellte multidimensionale Abfragesprache SQL-MDi, die erste Abfragesprache, welche die Integration von DWS derart umfassend unterstützt.

Kapitel 5

SQL-MDi und DA/FA-Algebra

Die multidimensionale Abfragesprache SQL-MDi [BS06a] wurde für den Einsatz in einem FDWS (vgl. Abschnitt 3.3.1) konzipiert und erlaubt die Auflösung der im Abschnitt 3.3.3 diskutierten Heterogenitäten auf Schema- und Instanzebene, welche bei der Integration von DWS auftreten können.

Der folgende Abschnitt stellt eine kurze, allgemeine Erläuterung von SQL-MDi und der zugrunde liegenden Fact-Algebra (FA) bzw. Dimension-Algebra (DA) dar. Im Abschnitt 5.2 werden die Syntaxelemente von SQL-MDi beispielhaft erläutert um die im Rahmenbeispiel vorhandenen Schema- (vgl. Abbildung 3.4) und Instanzkonflikte (vgl. Tabellen 3.2 und 3.3) zu bereinigen. Im Abschnitt 5.3 folgt eine detaillierte Erläuterung der den einzelnen SQL-MDi-Anweisungen zugrunde liegenden Algebra-Operatoren sowie deren Komposition in Form einer Operator-Baumstruktur. Eine Gegenüberstellung der im Abschnitt 5.2 vorgestellten SQL-MDi-Anweisungen zu den im Abschnitt 5.3 erläuterten DA/FA-Operatoren erfolgt im Abschnitt 5.4. Der letzte Abschnitt 5.5 stellt eine Zusammenfassung des Kapitels dar.

5.1 Theoretische Konzepte

SQL-MDi basiert auf dem „global as view“-Ansatz (vgl. Abschnitte 3.1.2 und 3.1.3), da es die Definition eines globalen DW-Schemas bzw. eines globalen Würfels mittels Sichten auf die lokalen DW-Schemata ermöglicht und eine prozedurale Vorschrift für die Erstellung und Instanziierung eines globalen Würfels darstellt. In diesem Zusammenhang spricht man von einem so genannten „globalen, instanziierten Würfel“:

Definition 5.1. Ein *globaler, instanziierter Würfel* entspricht einem globalen DW-Schema mit Instanzen bzw. Fakten (vgl. Definition 2.1). Er repräsentiert ein integriertes Schema der Komponenten-Würfel mitsamt deren Faktmengen.

Eine wesentliche zugrunde liegende Design-Entscheidung ist, dass SQL-MDi nur als Prolog für eine OLAP-Abfrage fungiert, d.h. eine SQL-MDi-Abfrage dient der Generierung und Instanziierung des globalen Würfels (jedoch ohne ihn physisch zu speichern), indem Mappings (vgl.

Definition 3.5) für die lokalen Würfel definiert werden. Das eigentliche Abfrageergebnis wird mit einer OLAP-Abfrage (z.B. mit SQL) berechnet, welche auf den globalen, instanziierten Würfel abgesetzt wird. [BS06a]

Die Dimension-Algebra (DA) und Fact-Algebra (FA) bilden die formale Basis von SQL-MDi. Mit DA-Ausdrücken werden Mappings definiert um lokale, heterogene Dimensionstabellen zu integrieren; mit FA-Ausdrücken werden Mappings definiert um autonome Fakttabellen zu integrieren. Die DA/FA stützt sich auf ein kanonisches, multidimensionales Datenmodell, welches die konzeptuellen Entitäten eines Data Warehouse bzw. Data Mart, nämlich Fakten und Dimensionen, formalisiert. Die DA/FA wurde für den Einsatz in der Föderations-Schicht der FDWS-Architektur der Abbildung 3.3 konzipiert. Mit der DA können die zwischen Export-Dimensions-Schemata bestehenden Heterogenitäten beseitigt und konfliktbereinigte Import-Dimensions-Schemata erzeugt werden, welche die Basis für die Generierung globaler Dimensionen bilden. Die globalen Dimensionen werden im „Dimension-Repository“ gespeichert. Mit der FA können die zwischen Export-Fakt-Schemata bestehenden Heterogenitäten beseitigt und konfliktbereinigte Import-Fakt-Schemata erzeugt werden. Aus den globalen Dimensionen des „Dimension-Repository“ und den Import-Fakt-Schemata wird das globale Schema des FDWS (= föderiertes Schema) konstruiert. [BS08]

SQL-MDi und die DA/FA basieren auf einem kanonischen, multidimensionalen Datenmodell (vgl. Abschnitt 5.3.2), welches die Modellierung von Fakten und Dimensionen unabhängig von der DW-Implementierung unterstützt. Dadurch können sowohl ROLAP- als auch MOLAP-Systeme (vgl. Abschnitt 2.4.2) integriert werden. Entsprechend der für diese Diplomarbeit getroffenen Annahme sind für die Entwicklung der Parser-Komponente für SQL-MDi nur ROLAP-Systeme von Bedeutung.

5.2 SQL-MDi-Syntax

Im Zuge der Arbeit an dieser Diplomarbeit wurden einige der in [BS06a] beschriebenen SQL-MDi-Syntaxelemente geändert, erweitert bzw. entfernt, da sie sich für die Implementierung der Parser-Komponente nur in anderer Form oder nur mit Erweiterungen als zweckmäßig erwiesen bzw. nicht erforderlich sind.

Das Grundgerüst einer SQL-MDi-Abfrage besteht aus folgenden Anweisungen:

```
DEFINE [GLOBAL] CUBE
MERGE DIMENSIONS
MERGE CUBES
```

Die `DEFINE [GLOBAL] CUBE` Anweisung wird zur Deklaration der in den globalen Würfel zusammenzuführenden lokalen Würfel verwendet. Mit einer Menge von Sub-Anweisungen können Heterogenitäten zwischen den lokalen Würfel-Schemata sowie zwischen den Instanzen der lokalen

Würfel (= Fakten) aufgelöst werden. Mit der `MERGE DIMENSIONS` Anweisung können zwischen lokalen Dimensionen bestehende Schema- und Instanzkonflikte mit diversen Sub-Anweisungen bereinigt werden und die lokalen Dimensionen in eine globale Dimension überführt werden. Alle konsolidierten Dimensionen, welche als globale Dimensionen in den globalen Würfel integriert werden sollen, müssen mit der `MERGE DIMENSIONS` Anweisung zusammengeführt werden. Die `MERGE CUBES` Anweisung dient der endgültigen Zusammenführung der lokalen, konsolidierten Würfel und der Berechnung des globalen, instanziierten Würfels. In diesem Zusammenhang ist u.a. zu spezifizieren, wie überlappende bzw. disjunkte Fakten (vgl. Abschnitt 3.3.3) gehandhabt werden sollen.

Die folgenden Ausführungen beschreiben, mit welchen Anweisungen welche Heterogenitäten beseitigt werden können. Dabei werden, entsprechend der Kategorisierung der Tabelle 3.1, zunächst die Anweisungen für die Eliminierung von Konflikten auf Schemaebene erläutert (inklusive der Konflikte im Schema-Instanz-Kontext), anschließend jene für die Eliminierung von Konflikten auf Instanzebene. Die vollständige Syntax-Spezifikation für SQL-MDi ist als EBNF-Grammatik im Anhang A zu finden.

5.2.1 Auflösung von Konflikten auf Schemaebene

Die in diesem Abschnitt präsentierten SQL-MDi-Anweisungen beziehen sich auf die lokalen DW-Schemata der Abbildung 3.4. Für jene Konflikte, welche in dieser Abbildung nicht vorhanden sind, werden weitere Annahmen getroffen um entsprechende SQL-MDi-Anweisungen formulieren zu können.

Dimensionalität
Heterogenitäten aufgrund unterschiedlicher Dimensionalität der zu integrierenden Würfel werden durch „Ausblenden“ der nicht in allen Würfeln vorhandenen Dimensionen aufgelöst. In SQL-MDi steht dafür die `DIM` Anweisung zur Verfügung, mit welcher jene dimensionalen Attribute einer Fakttabelle importiert werden, deren verbundene Dimensionen beibehalten werden sollen.

Beispiel 5.1: Das MFUB-Schema enthält eine zusätzliche Dimension „Werbeaktion“. Die einfachste Möglichkeit diesen Konflikt zu bereinigen besteht darin, diese Dimension durch ein implizites roll-up auf die Klassifikationsstufe „all“ „auszublenden“. Mit SQL-MDi kann dies folgendermaßen erreicht werden:

```
1 DEFINE CUBE mfuB::verkauf AS c2
2 (MEASURE c2.gespraechsdauer, MEASURE c2.umsatz,
3 DIM c2.datum, DIM c2.kunde, DIM c2.mobilnetz, DIM c2.umsatzkategorie)
```

Listing 5.1: SQL-MDi: Ausblenden einer Dimension

Zeile 1 des Listing 5.1 deklariert einen lokalen Würfel durch Angabe des DW-Knotens (mfuB) und der Fakttabelle (Verkauf) sowie eines Aliases (c2), welcher für die Referenzierung des deklarierten Würfels verwendet wird. Zeile 2 enthält zwei `MEASURE` Anweisungen, mit welchen die beiden Kenngrößen-Attribute der lokalen Fakttabelle importieren werden; Zeile 3 eine Menge von `DIM` Anweisungen, mit welchen die dimensionalen Attribute der Fakttabelle importiert werden. Wie im Listing 5.1 zu sehen, werden

alle dimensionalen Attribute bis auf „Werbeaktion“ importiert, wodurch die mit diesem verbundene Dimension ausgeblendet wird.

Unterschiedliche Aggregationshierarchien
Heterogenitäten aufgrund unterschiedlicher Aggregationshierachien in semantisch äquivalenten Dimensionen werden durch Entfernen jener Klassifikationsstufen, welche nicht in allen zu integrierenden Hierarchien vorhanden sind, aufgelöst. In SQL-MDi steht dafür die `MAP LEVELS` Anweisung zur Verfügung, in welcher jene Klassifikationsstufen einer Hierarchie angeführt werden, die beibehalten werden sollen.

Beispiel 5.2: Die beiden Dimensionen „Datum“ der lokalen DW-Schemata weisen unterschiedliche Aggregationshierarchien auf: im MFUB-Schema ist eine zusätzliche Klassifikationsstufe „Quartal“ vorhanden. Die einfachste Möglichkeit diesen Konflikt zu bereinigen besteht darin, die Klassifikationsstufe „Quartal“ zu entfernen. Mit SQL-MDi kann dies folgendermaßen erreicht werden:

```
1 DEFINE CUBE mfuB::verkauf AS c2
2 (MEASURE ..., DIM ..., DIM c2.datum
3     (MAP LEVELS c2.datum ([datum],[monat],[jahr]))
4 )
```

Listing 5.2: SQL-MDi: Löschen einer Klassifikationsstufe

Wie in Zeile 3 des Listing 5.2 ersichtlich, kann beim Import einer Dimension optional eine geschachtelte `MAP LEVELS` Anweisung formuliert werden, wo jene Klassifikationsstufen angeführt sind, welche beibehalten werden sollen, d.h. im obigen Listing wird die nicht-angeführte Klassifikationsstufe „Quartal“ gelöscht.

Namenskonflikte
Heterogenitäten aufgrund unterschiedlicher Namen für Kenngrößen-Attribute werden durch Umbenennen der Attribute aufgelöst. In SQL-MDi geschieht das mit einer `MEASURE` Anweisung in Kombination mit dem −> Operator.

Beispiel 5.3: Die Kenngrößen-Attribute „Dauer“ (MFUA-Schema) und „Gesprächsdauer“ (MFUB-Schema) sind semantisch äquivalent. Um den Namenskonflikt aufzulösen muss eines der beiden Kenngrößen-Attribute umbenannt werden. Mit SQL-MDi kann dies folgendermaßen erreicht werden:

```
1 DEFINE CUBE mufB::verkauf AS c2
2 (MEASURE c2.gespraechsdauer -> dauer, MEASURE c2.umsatz, ...)
```

Listing 5.3: SQL-MDi: Umbenennen eines Kenngrößen-Attributs

Im Listing 5.3 werden die beiden Kenngrößen-Attribute der lokalen Fakttabelle mit der `MEASURE` Anweisung importiert, wobei das Attribut „Gesprächsdauer“ mit dem −> Operator in „Dauer“ umbenannt wird.

Heterogenitäten aufgrund unterschiedlicher Namen für dimensionale Attribute einer Fakttabelle werden durch Umbenennen der Attribute aufgelöst. In SQL-MDi geschieht das mit einer `DIM` Anweisung in Kombination mit dem −> Operator.

Beispiel 5.4: Das dimensionale Attribut „svnr“ der Dimension „Kunde“ im MFUA-Schema heißt im MFUB-Schema „Kunde“. Um den Namenskonflikt aufzulösen, muss eines der beiden dimensionalen Attribute umbenannt werden. Mit SQL-MDi kann dies folgendermaßen erreicht werden:

```
1 DEFINE CUBE mfuB::verkauf AS c2
2 (...,DIM c2.kunde -> svnr, ...)
```

Listing 5.4: SQL-MDi: Umbenennen eines dimensionalen Attributs

Im Listing 5.4 wird das dimensionale Attribut „Kunde“ mit der `DIM` Anweisung importiert und mit dem −> Operator in „svnr“ umbenannt.

Heterogenitäten aufgrund unterschiedlicher Namen für Klassifikationsstufen werden durch Umbenennen der Attribute aufgelöst. In SQL-MDi geschieht das mit einer `MAP LEVELS` Anweisung in Kombination mit dem −> Operator.

Beispiel 5.5: Die Klassifikationsstufe „Monat“ der Dimension „Datum“ des MFUA-Schemas heißt im MFUB-Schema „Month“. Um den Namenskonflikt aufzulösen, muss eine der beiden Klassifikationsstufen umbenannt werden. Mit SQL-MDi kann dies folgendermaßen erreicht werden:

```
1 DEFINE CUBE mfuB::verkauf AS c2
2 (..., DIM c2.datum
3   (MAP LEVELS c2.datum ([datum],[month -> monat],[jahr])),
4 ...)
```

Listing 5.5: SQL-MDi: Umbenennen einer Klassifikationsstufe

Im Listing 5.5 wird die Klassifikationsstufe „Month“ innerhalb einer `MAP LEVELS` Anweisung mit dem −> Operator in „Monat“ umbenannt.

Heterogenitäten aufgrund unterschiedlicher Namen für nicht-dimensionale Attribute werden durch Umbenennen der Attribute aufgelöst. In SQL-MDi geschieht das mit einer `MATCH ATTRIBUTES` Anweisung.

Beispiel 5.6: Das nicht-dimensionale Attribut „Bezeichnung“ der Klassifikationsstufe „Tarif“ der Dimension „Kunde“ des MFUA-Schemas lautet im MFUB-Schema auf „Name“ und muss daher umbenannt werden. Mit SQL-MDi kann dies folgendermaßen erreicht werden:

```
1 DEFINE CUBE mfuA::verkauf AS c1
2 ...
3 DEFINE CUBE mfuB::verkauf AS c2
4 ...
5 DEFINE GLOBAL CUBE dw0::verkauf AS c0
6 MERGE DIMENSIONS c1.kunde AS d1, c2.kunde AS d2 INTO c0.kunde AS d0
7 (MATCH ATTRIBUTES d1.bezeichnung IS d2.name)
8 ...
```

Listing 5.6: SQL-MDi: Umbenennen eines nicht-dimensionalen Attributs

Die `MATCH ATTRIBUTES` Anweisung des Listing 5.6 bewirkt eine Umbenennung des Attributs „Name“ in den Namen des Attributs „Bezeichnung“. Im Unterschied zu den bisher behandelten Anweisungen ist `MATCH ATTRIBUTES` nicht einer `DEFINE CUBE` Anweisung untergeordnet, sondern einer `MERGE DIMENSIONS` Anweisung zur Zusammenführung der lokalen Dimensionen „Kunde“ in eine globale Dimension.

Domänenkonflikte
Heterogenitäten aufgrund unterschiedlicher Domänen für Kenngrößen-Attribute werden durch Einbinden einer Umrechnungsfunktion aufgelöst. In SQL-MDi steht dafür die `CONVERT MEASURES APPLY` Anweisung zur Verfügung.

Beispiel 5.7: Die Umsatz-Kenngrößen-Attribute der beiden lokalen DW-Schemata werden in unterschiedlichen Währungen gespeichert: die beiden Umsatz-Kenngrößen-Attribute im MFUA-Schema werden in Euro ausgewiesen, das Umsatz-Kenngrößen-Attribut im MFUB-Schema in US-Dollar. SQL-MDi bietet die Möglichkeit ein Kenngrößen-Attribut mit einer zuvor definierten Funktion zu konvertieren:

```
1 DEFINE CUBE mfuB::verkauf AS c2
2 ...
3 (CONVERT MEASURES APPLY usd2Eur() FOR c2.umsatz DEFAULT)
```

Listing 5.7: SQL-MDi: Konvertierung eines Kenngrößen-Attributs

Zeile 3 des Listing 5.7 zeigt eine `CONVERT MEASURES APPLY` Anweisung, mit welcher eine Währungsumrechnung für das Kenngrößen-Attribut „Umsatz“ des MFUB-Schemas mit der Funktion `usd2Eur()` erfolgt. Die Umrechnungsfunktion `usd2Eur()` muss dafür im System implementiert sein. Das Schlüsselwort `DEFAULT` drückt aus, dass keine einschränkenden Bedingungen gelten (ansonsten Schlüsselwort `WHERE` + Prädikat(e)), d.h. dass alle Umsatzwerte konvertiert werden sollen.

Heterogenitäten aufgrund unterschiedlicher Domänen für innere Klassifikationsstufen einer Hierarchie werden durch Entfernen der heterogenen Klassifikationsstufen aufgelöst. In SQL-MDi steht dafür die `MAP LEVELS` Anweisung zur Verfügung.

Beispiel 5.8: Die zusätzliche Klassifikationsstufe „Quartal“ der Dimension „Datum“ im MFUB-Schema bewirkt inkompatible roll-up Hierarchien, da eine zusätzliche Klassifikationsstufe zumindest eine weitere roll-up Beziehung erfordert [BG04]. Dieser Konflikt kann wiederum durch Löschen der Klassifikationsstufe „Quartal“ bereinigt werden (vgl. Listing 5.2).

Heterogenitäten aufgrund unterschiedlicher Domänen für unterste Klassifikationsstufen einer Hierarchie werden durch ein roll-up auf die nächste Klassifikationsstufe, deren Domäne kompatibel ist, aufgelöst. Diese Klassifikationsstufe wird zur neuen Basis-Klassifikationsstufe. In SQL-MDi steht dafür die `ROLLUP` Anweisung zur Verfügung.

Beispiel 5.9: Angenommen die Dimension „Datum“ des MFUA-Schemas enthält als unterste Klassifikationsstufe „Datum_std“, welche das Tagesdatum mit einer Zeitangabe in Stunden erfasst. Um nun kompatible Datum-Dimensionen zu erhalten, muss für das MFUA-Schema ein roll-up des mit dieser Klassifikationsstufe verknüpften dimensionalen Attributs der Fakttabelle auf „Datum“ erfolgen. Mit SQL-MDi kann dies folgendermaßen erreicht werden:

```
1 DEFINE CUBE mfuA::verkauf AS c1
2 ...
3 (ROLLUP c1.datum_std TO LEVEL c1.datum[datum])
```

Listing 5.8: SQL-MDi: Roll-up eines dimensionalen Attributs

Zunächst wird im Listing 5.8 mit `ROLLUP` das konfliktäre dimensionale Attribut der Fakttabelle angegeben, mit `TO LEVEL` wird die gewünschte neue Basis-Klassifikationsstufe angeführt. Alternativ könnte man die Klassifikationsstufe „Datum_std“ auch mit einer `MAP LEVELS` Anweisung löschen (vgl. Listing 5.2), da systemintern derselbe Operator zugrunde liegt.

Heterogenitäten aufgrund unterschiedlicher Domänen für nicht-dimensionale Attribute werden durch Einbinden einer Umrechnungsfunktion aufgelöst. In SQL-MDi steht dafür die `CONVERT ATTRIBUTES APPLY` Anweisung zur Verfügung.

Beispiel 5.10: Das nicht-dimensionale Attribut „Grundgebühr“, welches in der Dimension „Kunde“ der Klassifikationsstufe „Tarif“ zugeordnet ist, wird im MFUA-Schema in Euro ausgewiesen, im MFUB-Schema in US-Dollar. Analog zu den Domänenkonflikten bei Kenngrößen-Attributen (vgl. Beispiel 5.7) kann in SQL-MDi eine Funktion zur Konvertierung von nicht-dimensionalen Attributen und Klassifikationsstufen angegeben werden:

```
1 DEFINE CUBE mfuA::verkauf AS c1
2 ...
3 DEFINE CUBE mfuB::verkauf AS c2
4 ...
5 DEFINE GLOBAL CUBE dw0::verkauf AS c0
6 MERGE DIMENSIONS c1.kunde AS d1, c2.kunde AS d2 INTO c0.kunde AS d0
7 (CONVERT ATTRIBUTES APPLY usd2Eur() FOR d2.grundgebuehr DEFAULT)
8 ...
```

Listing 5.9: SQL-MDi: Konvertierung eines nicht-dimensionalen Attributs

Zeile 7 des Listing 5.9 zeigt eine `CONVERT ATTRIBUTES APPLY` Anweisung, mit welcher mit der Funktion `usd2Eur()` eine Währungsumrechnung für das nicht-dimensionale Attribut „Grundgebühr“ erfolgt. Die Umrechnungsfunktion `usd2Eur()` muss dafür im System implementiert sein. Wie `MATCH ATTRIBUTES` ist `CONVERT ATTRIBUTES APPLY` einer `MERGE DIMENSIONS` Anweisung zugeordnet.

Schema-Instanz-Konflikte

Heterogenitäten aufgrund von Schema-Instanz-Konflikten zwischen Fakttabellen treten auf, wenn in einem Schema der Kontext einer Kenngröße von einer Instanz eines dimensionalen Attributs bestimmt wird und in anderen Schemata separate Kenngrößen-Attribute für die unterschiedlichen Kontexte vorhanden sind. Um derart heterogene Fakttabellen zu integrieren, müssen die Fakttabellen-Schemata transformiert werden, wofür zwei Varianten möglich sind:

1. Transformation mehrerer Kenngrößen-Attribute in ein einzelnes, für dessen Werte der Kontext durch Instanzen eines neuen dimensionalen Attributs festgelegt wird.
   (im Rahmenbeispiel: Transformation MFUA → MFUB)
2. Transformation eines Kenngrößen-Attributs und eines dimensionalen (Kontext-)Attributs in mehrere separate Kenngrößen-Attribute für die einzelnen Instanzen des dimensionalen Attributs.
   (im Rahmenbeispiel: Transformation MFUB → MFUA)

Beispiel 5.11: Im MFUA-Schema ist je ein Kenngrößen-Attribut für Telefonie- und sonstige Umsätze vorhanden, wogegen im MFUB-Schema nur ein Kenngrößen-Attribut für Umsätze existiert, deren Kontext durch Instanzen des dimensionalen Attributs „Umsatzkategorie“ festgelegt wird (z.B.: die Fakttabelle „mfuB::Verkauf“ enthält ein Tupel mit einem Umsatz von 2.55, welcher aufgrund der zugeordneten Umsatzkategorie, „Umsatz_Telefonie“, als Telefonie-Umsatz zu interpretieren ist; vgl. Tabelle 3.3). Für die Auflösung des Schema-Instanz-Konflikts nach der Variante 1 steht in SQL-MDi die `PIVOT MEASURES` Anweisung zur Verfügung:

```
1 DEFINE CUBE mfuA::verkauf AS c1
2 (MEASURE c1.umsatz_telefonie, MEASURE c1.umsatz_sonstiges,
3 DIM c1.datum, DIM c1.svnr -> kunde, DIM c1.mobilnetz,
4 PIVOT MEASURES c1.umsatz_telefonie, c1.umsatz_sonstiges INTO c1.umsatz
5     USING c1.umsatzkategorie)
6 ...
```

Listing 5.10: SQL-MDi: Pivot-Variante 1

In den Zeilen 4 - 5 des Listing 5.10 werden in der `PIVOT MEASURES` Anweisung zunächst die Kenngrößen-Attribute des Quellschemas angeführt, mit `INTO` das neue, einzelne Kenngrößen-Attribut und mit `USING` das neue dimensionale (Kontext-)Attribut.

Beispiel 5.12: Für die Auflösung des Schema-Instanz-Konflikts des Beispiels 5.11 nach Variante 2 steht in SQL-MDi die `PIVOT MEASURE` Anweisung zur Verfügung:

```
1 DEFINE CUBE mfuB::verkauf AS c2
2 (MEASURE c2.umsatz,
3 DIM c2.datum, DIM c2.svnr, DIM c2.mobilnetz, DIM c2.umsatzkategorie,
4 PIVOT MEASURE c2.umsatz BASED ON c2.umsatzkategorie)
5 ...
```

Listing 5.11: SQL-MDi: Pivot-Variante 2

Im Listing 5.11 wird in einer `PIVOT MEASURE` Anweisung zunächst das Kenngrößen-Attribut des Quellschemas angeführt, mit `BASED ON` das dimensionale (Kontext-)Attribut.

Die Listings 5.10 und 5.11 enthalten keine `MEASURE` Anweisung zum Import des Kenngrößen-Attributs „Dauer“ bzw. „Gesprächsdauer“, wodurch diese Kenngrößen-Attribute aus den Fakttabellen entfernt werden.

5.2.2 Auflösung von Konflikten auf Instanzebene

Die in diesem Abschnitt präsentierten SQL-MDi-Anweisungen beziehen sich auf die lokalen DW-Instanzen der Tabellen 3.2 und 3.3.

Überlappende/Disjunkte Fakten
Überlappende Fakten bezeichnen Fakten mit identischen Primärschlüsseln, welche entweder in einer Kontext- oder Identitätsbeziehung zueinander stehen (vgl. Abschnitt 3.3.3). Abhängig von der Art der Beziehung erlaubt SQL-MDi die Integration der Faktmengen mit folgenden Konstrukten:

1. Sich aufgrund einer Kontextbeziehung überlappende Fakten können mit einer `AGGREGATE MEASURE` oder einer `TRACKING SOURCE AS DIMENSION` Anweisung integriert werden (vgl. Beispiele 5.13 und 5.14). Die `AGGREGATE MEASURE` Anweisung aggregiert die Werte der Kenngrößen-Attribute für die sich überlappenden Fakten; die `TRACKING SOURCE AS DIMENSION` Anweisung führt eine zusätzliche Kontextdimension im globalen Würfel ein, mit welcher für jede Kenngröße das Quell-DW identifiziert werden kann.
2. Sich aufgrund einer Identitätsbeziehung überlappende Fakten können mit einer `PREFER` Anweisung integriert werden (vgl. Beispiel 5.15). Mit der `PREFER` Anweisung können die Werte der Kenngrößen-Attribute eines Fakts bevorzugt werden, wodurch nur diese in den globalen Würfel übernommen werden.

Beispiel 5.13: Die Fakttabellen „Verkauf“ von MFUA und MFUB speichern jeweils einen Fakt mit dem Primärschlüssel {01.01.2006, 1234010180, MobCom}. Diese beiden Fakten stehen in einer Kontextbeziehung zueinander und können daher in SQL-MDi mit einer `AGGREGATE MEASURE` Anweisung integriert werden:

```
1  DEFINE CUBE mfuA::verkauf AS c1
2  ...
3  DEFINE CUBE mfuB::verkauf AS c2
4  ...
5  DEFINE GLOBAL CUBE dw0::verkauf AS c0
6  MERGE DIMENSIONS
7  ...
8  MERGE CUBES c1, c2 INTO c0 ON datum, kunde, mobilnetz, umsatzkategorie
9  (AGGREGATE MEASURE dauer IS SUM OF dauer DEFAULT,
10 AGGREGATE MEASURE umsatz IS SUM OF umsatz DEFAULT)
11 ...
```

Listing 5.12: SQL-MDi: Aggregation von überlappenden Fakten

Die Zeilen 1 - 7 des Listing 5.12 dienen der Deklaration von Würfeln und Dimensionen sowie der Auflösung bestehender Heterogenitäten. Die `MERGE CUBES` Anweisung besagt, dass die lokalen Würfel c1 und c2 in den globalen Würfel c0 auf Basis der dimensionalen Attribute „Datum“, „Kunde“, „Mobilnetz“ und „Umsatzkategorie“, welche auf einen gleichen Namen lauten müssen (d.h. ggf. zuvor umbenannt werden müssen), zusammengeführt werden sollen. Die beiden folgenden `AGGREGATE MEASURE` Anweisungen legen fest, dass die beiden globalen Kenngrößen „Umsatz“ und „Dauer“, bei überlappenden Fakten, aus der Summe der entsprechenden lokalen Kenngrößen berechnet werden. Anstatt SUM könnte auch MAX, MIN oder AVG zur Aggregation verwendet werden. Die lokalen Kenngrößen-Attribute müssen hier wiederum auf einen gleichen Namen lauten (d.h. sie müssen ggf. zuvor umbenannt werden). Das Schlüsselwort `DEFAULT` drückt aus, dass keine einschränkenden Bedingungen gelten (ansonsten Schlüsselwort `WHERE` + Prädikat(e)), d.h. dass alle Umsatz-Werte bzw. Dauer-Werte mit identischen Koordinaten aggregiert werden.

Beispiel 5.14: Die überlappenden Fakten des Beispiels 5.13 können in SQL-MDi auch mit einer `TRACKING SOURCE AS DIMENSION` Anweisung integriert werden:

```
1  DEFINE CUBE mfuA::verkauf AS c1
2  ...
3  DEFINE CUBE mfuB::verkauf AS c2
4  ...
5  DEFINE GLOBAL CUBE dw0::verkauf AS c0
6  MERGE DIMENSIONS
7  ...
8  MERGE CUBES c1, c2 INTO c0 ON datum, kunde, mobilnetz, umsatzkategorie
9  (TRACKING SOURCE AS DIMENSION source(VARCHAR(10))
10     IS 'mfuA' WHERE SOURCE()='c1', IS 'mfuB' WHERE SOURCE()='c2')
11 ...
```

Listing 5.13: SQL-MDi: Kontextdimension für überlappende Fakten

Im Unterschied zum Listing 5.12 wird im Listing 5.13 eine Kontextdimension zur Handhabung der Kenngrößen verwendet. Die `TRACKING SOURCE AS DIMENSION` Anweisung erlaubt das Inkludieren einer zusätzlichen Dimension in den globalen Würfel um für jeden Wert eines Kenngrößen-Attributs das Quell-DW identifizieren zu können (`WHERE SOURCE()` + Prädikat).

Beispiel 5.15: Um zu demonstrieren, wie sich aufgrund einer Identitätsbeziehung überlappende Fakten integriert werden können, wird für dieses Beispiel angenommen, dass die Fakten des Beispiels 5.13 in einer Identitätsbeziehung zueinander stehen. Eine solche Beziehung wird mit einer `PREFER` Anweisung behandelt:

```
1  DEFINE CUBE mfuA::verkauf AS c1
2  ...
3  DEFINE CUBE mfuB::verkauf AS c2
4  ...
5  DEFINE GLOBAL CUBE dw0::verkauf AS c0
6  MERGE DIMENSIONS
7  ...
8  MERGE CUBES c1, c2 INTO c0 ON datum, kunde, mobilnetz, umsatzkategorie
9  (PREFER c1.dauer DEFAULT,
10 PREFER c1.umsatz DEFAULT)
11 ...
```

Listing 5.14: SQL-MDi: Bevorzugung von überlappenden Fakten

Im Listing 5.14 wird für die Bevorzugung von Fakten eines lokalen Würfels die `PREFER` Anweisung verwendet, mit welcher das zu bevorzugende Kenngrößen-Attribut (mit Angabe des lokalen Würfels) festgelegt wird. Das Schlüsselwort `DEFAULT` drückt aus, dass keine einschränkenden Bedingungen gelten (ansonsten Schlüsselwort `WHERE` + Prädikat(e)), d.h. dass bei allen Umsatz- bzw. Dauer-Werten mit identischen Koordinaten jene von c1 im globalen Würfel verwendet werden.

Für die Behandlung von disjunkten Fakten, d.h. Fakten mit unterschiedlichen Primärschlüsseln, bietet SQL-MDi optionale Schlüsselwörter für Mengenoperationen, wie UNION, MINUS, JOIN oder FULL OUTER JOIN (mit der Semantik der äquivalenten SQL-Mengenoperatoren (vgl. [KE04])) innerhalb der `MERGE CUBES` Anweisung. Wird keine Mengenoperation angegeben, so wird standardmäßig UNION verwendet.

Beispiel 5.16: Verwendet man das Schlüsselwort UNION innerhalb der `MERGE CUBES` Anweisung, werden alle Fakten in den globalen Würfel übernommen, wobei für die Handhabung von überlappenden Fakten die zuvor beschriebenen Anweisungen verwendet werden müssen:

```
1 ...
2 MERGE CUBES c1, c2 INTO c0 UNION ON datum, kunde, mobilnetz, umsatzkategorie
3 -- Anweisungen für überlappende Fakten
4 ...
```

Listing 5.15: SQL-MDi: Handhabung von disjunkten Fakten

Namenskonflikte

Heterogenitäten aufgrund unterschiedlicher Namen für Dimensionsinstanzen werden durch Umbenennung aufgelöst. In SQL-MDi geschieht das mit einer `RENAME` Anweisung.

Beispiel 5.17: Im MFUA-Schema enthält die Dimensionstabelle „Mobilnetz“ eine Instanz „HandyTel“, im MFUB-Schema eine Instanz „HandyTelInc“; beide beschreiben jedoch dasselbe Objekt der Wirklichkeit. Zur Lösung dieses Namenskonflikts muss eine Instanz auf die entsprechend andere umbenannt werden, was durch folgende SQL-MDi-Anweisung erfolgt:

```
1 ...
2 MERGE DIMENSIONS c1.mobilnetz AS d1, c2.mobilnetz AS d2 INTO c0.mobilnetz AS d3
3 (RENAME d1.mobilnetz >> 'HandyTelInc' WHERE c1.mobilnetz.mobilnetz='HandyTel')
4 ...
```

Listing 5.16: SQL-MDi: Umbenennen von Dimensionsinstanzen

Die `RENAME` Anweisung des Listing 5.16 benennt die Dimensionsinstanz „HandyTel“ (Schlüsselwort `WHERE`) des dimensionalen Attributs „Mobilnetz“ mit dem >> Operator in „HandyTelInc“ um. Sollten viele Dimensionsinstanzen umbenannt werden müssen, erlaubt die `RENAME` Anweisung das Einbinden einer Mapping-Tabelle, welche Instanzen eines dimensionalen Attributs auf Instanzen des entsprechenden Attributs anderer Dimensionen abbildet.

Überlappende/Disjunkte Dimensionsinstanzen

Analog zu überlappenden bzw. disjunkten Fakten muss auch für Dimensionsinstanzen entschieden werden, wie diese in die globalen Dimensionen integriert werden. Dafür bietet SQL-MDi optionale Schlüsselwörter für Mengenoperationen innerhalb der `MERGE DIMENSIONS` Anweisung. Wird keine Mengenoperation angegeben, so wird standardmäßig UNION verwendet.

Beispiel 5.18: Verwendet man das Schlüsselwort MINUS innerhalb der `MERGE DIMENSIONS` Anweisung für die Dimension „Mobilnetz“, so scheint nur die Instanz „FiveCom“ der Dimension „Mobilnetz“ des MFUA-Schemas in der globalen Dimension auf:

```
1 ...
2 MERGE DIMENSIONS c1.mobilnetz AS d1, c2.mobilnetz AS d2 INTO c0.mobilnetz AS d0 MINUS
3 ...
```

Listing 5.17: SQL-MDi: Überlappende/Disjunkte Dimensionsinstanzen

Heterogene roll-up Beziehungen

Heterogenitäten aufgrund heterogener roll-up Beziehungen werden durch Bevorzugung einer roll-up Beziehung einer Dimension aufgelöst. Dafür steht in SQL-MDi die `RELATE` Anweisung zur Verfügung.

Beispiel 5.19: Der Konflikt aufgrund der heterogenen roll-up Beziehungen 1234010180 ↦ SparCom (MFUA-Schema) und 1234010180 ↦ FullCom (MFUB-Schema) kann mit folgender SQL-MDi-Anweisung bereinigt werden:

```
1 ...
2 MERGE DIMENSIONS c1.kunde AS d3, c2.kunde AS d4 INTO c5.kunde AS d5
3 (RELATE d3.svnr, d4.kunde WHERE d3.svnr=d4.kunde USING HIERARCHY OF d3)
4 ...
```

Listing 5.18: SQL-MDi: Überschreiben einer roll-up Beziehung

Die RELATE Anweisung des Listing 5.18 bewirkt ein Überschreiben der roll-up Beziehung von d4.kunde mit der entsprechenden Beziehung von d3.svnr, d.h. '1234010180' ↦ 'FullCom' wird mit '1234010180' ↦ 'SparCom' überschrieben. Die bevorzugte roll-up Beziehung wird mit dem Schlüsselwort USING HIERARCHY OF und der Angabe des Alias der bevorzugten Dimension definiert.

5.2.3 Formulierung einer SQL-MDi-Abfrage

Nachdem nun die einzelnen Anweisungen zur Auflösung der unterschiedlichen Heterogenitäten vorgestellt wurden, wird in diesem Abschnitt erläutert, wie bei der Formulierung einer SQL-MDi-Abfrage am besten vorgegangen werden sollte.

1. **Erzeugung kompatibler Faktschemata:**
   Nachdem die semantisch äquivalenten Kenngrößen- und dimensionalen Attribute der lokalen Würfel identifiziert wurden, müssen diese mit MEASURE bzw. DIM Anweisungen für jeden Würfel importiert werden. Bestehen heterogene Dimensionshierarchien, so müssen die konfliktären Klassifikationsstufen in einer geschachtelten MAP LEVELS Anweisung entfernt werden. Mögliche Namenskonflikte bei den Kenngrößen- bzw. dimensionalen Attributen und Klassifikationsstufen werden mit dem −> Operator in der MEASURE Anweisung bzw. in der DIM oder MAP LEVELS Anweisung aufgelöst. Bei Anwendung des −> Operators besteht die Möglichkeit, dass in den folgenden Abfrageteilen sowohl der alte als auch der neue Name verwendet wird (ausgenommen in der MERGE CUBES Anweisung). Für mögliche Schema-Instanz-Konflikte definiert man PIVOT Anweisungen, wobei hier die neu entstehenden Namen für die Kenngrößen-Attribute berücksichtigt werden müssen, d.h. je nach gewählter Variante müssen Kenngrößen-Attribute bzw. Instanzen des dimensionalen (Kontext-)Attributs ggf. umbenannt werden um Namensgleichheit zu gewährleisten. Darüber hinaus sind bei Bedarf ROLLUP und CONVERT MEASURES APPLY Anweisungen zu definieren.
2. **Erzeugung kompatibler Dimensionsschemata und Zusammenführung der lokalen Dimensionen in globale Dimensionen:**
   Nachdem kompatible Faktschemata erzeugt wurden, können zwischen lokalen Dimensionen bestehende Konflikte auf Schema- und Instanzebene aufgelöst werden und die lokalen Dimensionen mit MERGE DIMENSIONS Anweisungen in globale Dimensionen überführt werden. Bei der Formulierung der MERGE DIMENSIONS Anweisung ist darauf zu achten, dass sich die Namen der Dimensionen von jenen der dimensionalen Attribute der Fakttabelle (vgl. DIM Anweisung) unterscheiden können.

3. **Zusammenführung der lokalen Würfel:**
   Als letzten Teil einer SQL-MDi-Abfrage definiert man eine `MERGE CUBES` Anweisung, in welcher mit diversen Sub-Anweisungen wie `PREFER` oder `TRACKING SOURCE AS DIMENSION` die Handhabung von überlappenden Fakten spezifiziert wird. Damit die `MERGE CUBES` Anweisung korrekt verarbeitet werden kann, müssen äquivalente Namen für Kenngrößen- und dimensionale Attribute bestehen. Am Ende einer SQL-MDi-Abfrage werden die globalen Kenngrößen- und dimensionalen Attribute, welche Teil des globalen Würfels sind, nochmals explizit angeführt (`MEASURE` bzw. `DIM` Anweisung). Hier können ggf. noch globale Kenngrößen- und dimensionale Attribute ausgeblendet werden.

Listing 5.19 zeigt eine vollständige SQL-MDi-Abfrage, welche die zuvor präsentierten Schema- und Instanzkonflikte des Rahmenbeispiels auflöst:

```
1  DEFINE
2  CUBE mfuA::verkauf AS c1
3  (MEASURE c1.dauer, MEASURE c1.umsatz_telefonie, MEASURE c1.umsatz_sonstiges,
4  DIM c1.datum_std, DIM c1.svnr, DIM c1.mobilnetz,
5  PIVOT MEASURES c1.umsatz_telefonie, c1.umsatz_sonstiges
6    INTO c1.umsatz USING c1.umsatzkategorie)
7  (ROLLUP c1.datum_std TO LEVEL c1.datum[datum])
8
9  CUBE mfuB::verkauf AS c2
10 (MEASURE c2.gespraechsdauer -> dauer, MEASURE c2.umsatz,
11 DIM c2.datum
12   (MAP LEVELS c2.datum([datum],[month -> monat],[jahr])),
13 DIM c2.kunde -> svnr
14   (MAP LEVELS c2.kunde([kunde -> svnr],[tarif]),
15 DIM c2.mobilnetz, DIM c2.umsatzkategorie)
16 (CONVERT MEASURES APPLY usd2Eur() FOR c2.umsatz DEFAULT)
17
18 GLOBAL CUBE dw0::verkauf AS c0
19
20 MERGE DIMENSIONS c1.datum AS d1, c2.datum AS d2 INTO c0.datum AS d0
21
22 MERGE DIMENSIONS c1.kunde AS d3, c2.kunde AS d4 INTO c0.kunde AS d5
23 (RELATE d3.svnr, d4.kunde WHERE d3.svnr=d4.kunde USING HIERARCHY OF d3)
24 (MATCH ATTRIBUTES d3.bezeichnung IS d4.name)
25 (CONVERT ATTRIBUTES APPLY usd2Eur() FOR d4.grundgebuehr DEFAULT)
26
27 MERGE DIMENSIONS c1.mobilnetz AS d6, c2.mobilnetz AS d7 INTO c0.mobilnetz AS d8
28 (RENAME d6.mobilnetz >> 'HandyTelInc' WHERE c1.mobilnetz='HandyTel')
29
30 MERGE DIMENSIONS c1.umsatzkategorie AS d9, c2.umsatzkategorie AS d10
31   INTO c0.umsatzkategorie AS d11
32
33 MERGE CUBES c1,c2 INTO c0 ON datum, svnr, mobilnetz, umsatzkategorie
34 AGGREGATE MEASURE dauer IS SUM OF dauer,
35 AGGREGATE MEASURE umsatz IS SUM OF umsatz
36 (MEASURE dauer, MEASURE umsatz, DIM datum, DIM svnr, DIM mobilnetz, DIM umsatzkategorie)
```

Listing 5.19: SQL-MDi: Vollständige SQL-MDi-Abfrage

Der folgende Abschnitt erläutert die SQL-MDi zugrunde liegende Algebra sowie deren Operatoren, aus welchen die im Zuge dieser Diplomarbeit entwickelte Parser-Kompontente eine Baumstruktur generiert.

5.3 Dimension-Algebra/Fact-Algebra

In diesem Abschnitt wird die formale Basis von SQL-MDi, nämlich die Dimension-Algebra (DA) bzw. Fact-Algebra (FA), erläutert [BS08]. Im folgenden Abschnitt wird zunächst der praktische Nutzen einer formalen Algebra für eine Abfragesprache dargestellt. Im Abschnitt 5.3.2 erfolgt eine Formalisierung des multidimensionalen Datenmodells, bevor die Operatoren der DA/FA sowie deren Komposition zu einer Baumstruktur in den Abschnitten 5.3.3 und 5.3.4 veranschaulicht werden.

5.3.1 Eine formale Algebra für SQL-MDi

Die Relationale Algebra als Grundlage von SQL unterstützt die Abfrageoptimierung auf logischer Ebene und die Implementierung von SQL [KE04]. Da diese Vorzüge auch für SQL-MDi wünschenswert sind, wurde die DA/FA als formale Basis von SQL-MDi konzipiert. Ein weiterer Beweggrund für die Entwicklung der DA/FA war die Gewährleistung der SQL-Kompatibilität von SQL-MDi, d.h. dass eine SQL-MDi-Abfrage in eine Menge von SQL-Anweisungen für die einzelnen Komponentensysteme transformiert wird. Die Komplexität dieser Transformation kann durch Implementierung einer Zwischensprache [Mös06a] in Form der DA/FA reduziert werden. Abbildung 5.1 veranschaulicht den dadurch vorgegebenen Verarbeitungsablauf für eine SQL-MDi-Abfrage.

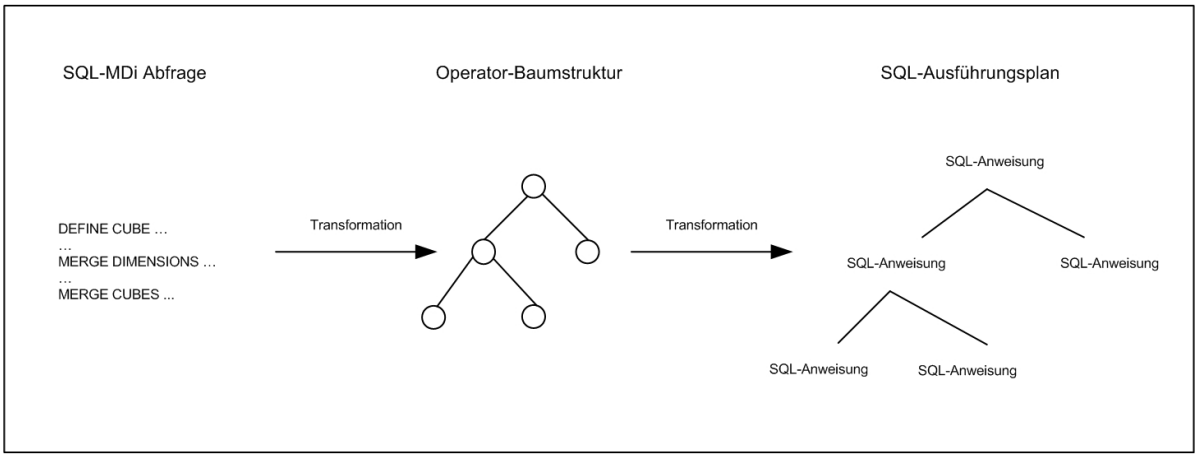


Abbildung 5.1: Verarbeitung einer SQL-MDi-Abfrage

Die SQL-MDi-Abfrage wird im ersten Schritt in eine Baumstruktur mit den DA/FA-Operatoren transformiert. Erst im zweiten Schritt erfolgt die Generierung einzelner SQL-Anweisungen, welche auf Basis der Operator-Baumstruktur in einem Ausführungsplan angeordnet werden. Die Aufgabe der im Zuge dieser Diplomarbeit entwickelten Parser-Komponente ist die Transformation einer SQL-MDi-Abfrage in eine Operator-Baumstruktur. Der zweite Transfor-

mationsschritt, die Generierung von SQL-Anweisungen aus einer Operator-Baumstruktur, ist Gegenstand einer weiteren, in Arbeit befindlichen Diplomarbeit [Ros08].

5.3.2 Kanonisches, multidimensionales Datenmodell

Das verwendete Datenmodell kennt die multidimensionalen Konstrukte der Fakten und Dimensionen unabhängig von einer bestimmten Implementierungsplattform. Dadurch können DW-Produkte unterschiedlicher Hersteller im FDWS integriert werden. In diesem Abschnitt wird das im Abschnitt 2.3 vorgestellte multidimensionale Datenmodell, auf welchem die DA/FA basiert, formalisiert [BS08]. Die formalen, mathematischen Definitionen wurden unverändert aus [BS08] übernommen.

Definition 5.2. Ein *Data Warehouse* $DW = \{C_1, ..., C_n, D_1, ..., D_m\}$ besteht aus einer nicht-leeren Menge von Würfeln C_i und einer nicht-leeren Menge von Dimensionen D_j.

Definition 5.3. Ein *Würfel* $C = [F_C, D_C]$ besteht aus einer Menge von Fakten bzw. Zellen F_C, welche mit einer Menge von Dimensionen $D_C \subseteq \{D_1, ..., D_m\}$ verbunden sind. D_C repräsentiert den multidimensionalen Kontext der Fakten F_C bzw. die Dimensionalität des Würfels C.

Für die folgenden Definitionen sei $\tau_1, ..., \tau_m$ eine endliche Menge von Datentypen (z.B. Integer), deren Domänen durch die Funktion $\mathrm{dom}(\tau)$ repräsentiert werden.

Definition 5.4. Eine *Dimension* $D \in \{D_1, ..., D_m\}$ ist definiert durch:

- das *Dimensionsschema* $S_D = (L_D, S(L_D), H_D)$ besteht aus (1) einer endlichen, nicht-leeren Menge von Klassifikationsstufen $L_D = \{l_1, ..., l_j, ..., l_m, l_{all}\}$ (2) mit dem Klassifikationsstufen-Schema $S(L_D) = \{S_{l1}, ..., S_{lm}\}$ und (3) einer Hierarchie $H_D \subseteq L_D \times L_D$, wobei H_D einen Verbund darstellt. Sei $l_1, l_2 \in H_D$, $l_1 \mapsto l_2$ bedeutet, dass l_1 „rolls-up to“ l_2.
- das *Klassifikationsstufen-Schema* $S_{lj} \in S(L_D)$ einer Klassifikationsstufe l_j ist ein Schema von Attributen $(k_j, a_{ji}, ..., a_{jk})$ mit dem Namen l_j, dem Schlüssel oder dimensionalen Attribut bzw. „roll-up Attribut“ k_j und den nicht-dimensionalen Attributen $a_{ji}, ..., a_{jk}$. Für die einem Attribut $A_k \in S_{lj}$ zugrunde liegende Domäne gilt, $dom(A_k) = dom(\tau_k)$.
- der *Dimensionsinstanz* $\mathrm{d}(S_D)$ über dem Schema S_D mit dem Namen d, welche (1) eine Menge von Instanzen V_d, die ein Tupel über dem Klassifikationsstufen-Schema L_D darstellen, und (2) eine Menge von „roll-up“ Beziehungen ρ_d zwischen Teilmengen $T_j \subseteq V_d$ (siehe Definition 5.6), enthält.

Definition 5.5. Die *Basis-Klassifikationsstufe* l_0 einer Hierarchie H_D repräsentiert die feinste Granularität und bildet das unterste Element von H_D. Die Klassifikationsstufe „all“ l_{all} repräsentiert die gröbste Granularität und ist das oberste Element von H_D.

Definition 5.6. Sei D eine Dimension mit dem Schema S_D, dem Klassifikationsstufen-Schema $S(L_D)$ und der Instanz $d(S_D)$, so können folgende Funktionen über D definiert werden:

- *level*: $V_d \mapsto L_D$ ermittelt die Klassifikationsstufe l_j für ein gegebenes $v_i \in V_d$.
- *members*: $L_D \mapsto 2^{V_d}$ ermittelt die Menge $T_i = \{v \in V_d | level(v) = l_i\}$, welche alle Elemente $v \in V_d$ der Klassifikationsstufe l_i umfasst.

Definition 5.7. Seien $l, k \in L_D$ zwei Klassifikationsstufen einer Dimension D, wobei $l \neq k$, and $T_l = members(l), T_k = members(k)$. Die roll-up Funktion $\rho^{l \mapsto k}$ ist für jedes Instanz-Paar in $l, k \in L_D$ definiert und konsistent, wenn $\forall v \in T_l : \rho^{l \mapsto k}(v) = w \wedge w \in T_k$. Die Menge der roll-up Funktionen ρ_d umfasst alle derart definierten $\rho^{l \mapsto k}$ (vgl. Definition 5.4).

Definition 5.8. Die *Fakten* bzw. *Zellen* F_C eines Würfels C sind definiert durch:

- das *Faktschema* $S_C = \{A_C, M_C\}$ mit (1) einer Menge von dimensionalen Attributen $A_C = \{A_1, ..., A_n\}$ und (2) einer Menge von Kenngrößen-Attributen $M_C = \{M_1, ..., M_m\}$. Jedes $A_i \in A_C$ ist mit einer Klassifikationsstufe $l_j \in L_D$ verknüpft, jedes $M_j \in M_C$ mit einer Domäne τ_j. Die Domäne der Attribute in S_C ist definiert als $dom(A_i) = members(l_i)$ und $dom(M_j) = dom(\tau_j)$.
- die *Faktinstanz* $c(S_C)$, eine Menge von Tupeln über $\{[dom(A_1) \times ... \times dom(A_n)], [dom(M_1) \times ... \times M_m]\}$. Ein Tupel $f \in c(S_C)$ wird als „Fakt“ oder „Zelle“ bezeichnet. Die Werte $[f(A_1), ..., f(A_n)]$ werden als Koordinaten einer Zelle bezeichnet und bilden den multidimensionalen Kontext für die Kenngrößen $[f(M_1), ..., f(M_m)]$.

Definition 5.9. Seien $f_1 \in c_1(S_{C_1})$ und $f_2 \in c_2(S_{C_2})$ zwei Faktmengen mit exakt denselben dimensionalen Attributen, d.h. $A_{C_1} = A_{C_2} = A$. Die Zellen f_1 und f_2 sind „überlappend“, wenn $f_1(A) = f_2(A)$.

Dieses formalisierte, multidimensionale Datenmodell ist ein konzeptuelles Modell, d.h. es kann auf unterschiedlichen Plattformen implementiert werden um eine bestimmte Abfragesprache auf das globale Schema zu unterstützen. Wird das globale Schema beispielsweise in einem relationalen System implementiert, so kann SQL für Abfragen auf den globalen Würfel verwendet werden, wobei SQL-MDi der Spezifikation der Mappings dient. [BS08] Im folgenden Abschnitt werden die Operatoren der Dimension- bzw. Fact-Algebra erläutert.

5.3.3 Operatoren der Dimension- und Fact-Algebra

Jeder Operator der DA/FA repräsentiert genau eine Berechnung auf einem Operanden, d.h. auf einer Dimension bzw. einem Würfel. Dadurch ist gewährleistet, dass eine vom Parser erzeugte Operator-Baumstruktur einfach zu verarbeiten ist. In diesem Abschnitt werden die Operatoren allgemein beschrieben und deren Anwendung anhand des Rahmenbeispiels der Abbildung 3.4 bzw. der Tabellen 3.2 und 3.3 demonstriert. Darüber hinaus werden die dem jeweiligen Operator zugeordnete(n) SQL-MDi-Anweisung(en) angeführt (vgl. zusätzlich Abschnitt 5.4).

5.3.3.1 Operatoren der Dimension-Algebra

Die Gruppe dieser Operatoren ist dadurch gekennzeichnet, dass sie eine Operation auf genau einer lokalen Dimension repräsentieren. Werden DA-Operatoren auf eine Input-Dimension D angewendet, so wird eine Output-Dimension D' erzeugt, in welcher das ursprüngliche Schema S_D und/oder deren Instanzen $d(S_D)$ modifiziert wurden. Im Folgenden werden die in der Parser-Komponente implementierten DA-Operatoren in Anlehnung an [BS08] beschrieben.

Rename: ζ.
Mit diesem Operator können Namen von Attributen in d' umbenannt werden, wobei ζ auf folgende Objekte angewendet werden kann:

- Umbenennen einer Klassifikationsstufe $l \in L_D : \zeta_{l' \leftarrow l}(d)$.
- Umbenennen eines Attributs $l.a$ eines Klassifikationsstufen-Schemas $S_l \in S(L_D)$: $\zeta_{a' \leftarrow l.a}(d)$.

Beispiel 5.20: $\zeta_{monat \leftarrow month}(datum)$, angewendet auf die Dimension „mfuB::verkauf.datum“, bewirkt eine Umbenennung der Klassifikationsstufe „Month“ in „Monat“.

Zugeordnete SQL-MDi-Anweisungen:
`MAP LEVELS`, `MATCH ATTRIBUTES`

Change: $\delta_{w \leftarrow v(l.k_j|l.a_i)}(d)$.
Mit diesem Operator können Elemente aus V_d geändert werden, d.h. Werte eines dimensionalen Attributs k_j oder eines nicht-dimensionalen Attributs a_i. $\delta_{w \leftarrow v(l.k_j|l.a_i)}(d)$ bewirkt eine Änderung des Werts v eines dimensionalen Attributs k_j oder eines nicht-dimensionalen Attributs $a.i$ auf den Wert w.

Beispiel 5.21: $\delta_{'HandyTelInc' \leftarrow 'HandyTel'(mobilnetz.mobilnetz)}(mobilnetz)$, angewendet auf die Dimension „mfuB:verkauf.mobilnetz“, bewirkt eine Umbenennung des Werts „HandyTel“ des dimensionalen Attributs „Mobilnetz“ der Klassifikationsstufe „Mobilnetz“ in „HandyTelInc“.

Zugeordnete SQL-MDi-Anweisungen:
`RENAME`, `RENAME CONTEXT DIM`

Convert: $\gamma_{(l.k_j|l.a_i)\theta}(d)$.
Mit diesem Operator kann die Domäne eines dimensionalen Attributs k_j oder eines nicht-dimensionalen Attributs a_i einer Klassifikationsstufe l_j geändert werden, indem eine Funktion θ auf $dom(k_j)$ bzw. $dom(a_i)$ angewendet wird.

Beispiel 5.22: $\gamma_{tarif.grundgebuehr\ usd2Eur()}(kunde)$, angewendet auf die Dimension „mfuB:verkauf.kunde“, bewirkt eine Konvertierung des nicht-dimensionalen Attributs „Grundgebühr“ der Klassifikationsstufe „Tarif“ von US-Dollar in Euro.

Zugeordnete SQL-MDi-Anweisung:
`CONVERT ATTRIBUTES APPLY`

Delete level: $\alpha_{l_j}(d)$.
Mit diesem Operator kann eine Klassifikationsstufe einer Dimension aus allen Hierarchien, in denen sie verwendet wird, gelöscht werden (ausgenommen die Klassifikationsstufe „all“). Als Parameter erfordert dieser Operator die zu löschende Klassifikationsstufe l_j. Die Anwendung von $\alpha(l_j)$ beeinflusst H_D und V_d/ρ_d folgendermaßen:

- H_D:
  - Ist l_j nicht die Basis-Klassifikationsstufe der Hierarchie, so werden die roll-up Beziehungen $l_{j-1} \mapsto l_j$ und $l_j \mapsto l_{j+1}$ gelöscht und ein neues roll-up $l_{j-1} \mapsto l_{j+1}$ wird erzeugt.
  - Ist l_j die Basis-Klassifikationsstufe der Hierarchie, so wird nur $l_j \mapsto l_{j+1}$ gelöscht. In diesem Fall muss jedoch zusätzlich S_C angepasst werden, indem $A_i \in S_C$ mit l_{j+1} ersetzt wird.
- V_d/ρ_d:
  - Ist l_j nicht die Basis-Klassifikationsstufe der Hierarchie, so werden alle Instanzen T_j gelöscht; die roll-up Beziehungen werden analog zu S_D geändert, indem der transitive Charakter der Beziehungen genutzt wird: $\{\forall m \in members(l_{j-1}) : \rho(\rho(m)) = \rho^{l_{j-1} \mapsto l_{j+1}}(m)\}$
  - Ist l_j die Basis-Klassifikationsstufe der Hierarchie, so werden alle Instanzen T_j gelöscht, wobei in diesem Fall $c(S_C)$ modifiziert werden muss, indem $\forall f(A_i) \in c(S_C) : \rho^{l_j \mapsto l_{j+1}}(f(A_i))$, wobei $f(A_i) \in members(l_j)$.

Beispiel 5.23: $\alpha_{quartal}(datum)$, angewendet auf die Dimension „mfuB:verkauf.datum“, bewirkt ein Löschen der Klassifikationsstufe „Quartal“ der Dimension „Kunde“. Da „Quartal“ keine Basis-Klassifikationsstufe darstellt, wird H_D derart modifiziert, dass Month $\mapsto$ Jahr und Vd/ρ_d derart, dass $\forall m \in members(month) : \rho(\rho(m)) = \rho^{month \mapsto jahr}(m)$.

Zugeordnete SQL-MDi-Anweisungen:
`MAP LEVELS`, `ROLLUP`

Override rollup: $\Omega_{m \mapsto v}(d)$.
Dieser Operator ändert die roll-up Beziehung des Elements m auf $m \mapsto v$, wobei $m, v \in V_d$, $l_m = level(m)$, $l_v = level(v)$, $l_m \mapsto l_v \in H_D$. Die Anwendung von $\Omega_{m \mapsto v}(d)$ bewirkt eine Änderung des Ergebnisses von $\rho^{l_m \mapsto l_v}(m)$ auf v.

Beispiel 5.24: $\Omega_{'12334010180' \mapsto 'FullCom'}(kunde)$, angewendet auf die Dimension „mfuA:verkauf.kunde“, bewirkt ein Überschreiben der roll-up Beziehung von '1234010180' $\mapsto$ 'SparCom' mit '1234010180' $\mapsto$ 'Fullcom'.

Zugeordnete SQL-MDi-Anweisung:
`RELATE`

Algorithmus: mergeDim
Input: $DW_1.D, ..., DW_n.D$.
Output: $DW.D$.
begin
1 $DW.L_D = \bigcup_{i=1..n} DW_i.L_D$;
2 $DW.H_D = \bigcup_{i=1..n} DW_i.H_D$;
3 $DW.V_d = \bigcup_{i=1..n} members(DW_i.V_d)$;
4 $\forall l, k \in DW.H_D : DW.\rho_d = DW.\rho_d \cup \rho^{l \mapsto k}$;
end mergeDim;

Um die globalen Dimensionen des „Dimension-Repository“ aus den, mit DA-Ausdrücken definierten, Import-Dimensions-Schemata zu erzeugen, wird der Algorithmus `mergeDim` angewendet. Angenommen $DW_i.D$ repräsentiert die Dimension D des DW_i im FDWS. Der Algorithmus berechnet aus einer Menge von n Dimensionen D mit identischen Namen, aus n unterschiedlichen DW, eine einzelne Output-Dimension $DW.D$ mit der Menge von Klassifikationsstufen $DW.L_D$, der Hierarchie $DW.H_D$, der Menge von Dimensionsinstanzen $DW.V_D$ sowie der Menge von roll-up Beziehungen $DW.\rho_d$. Das Ergebnis von `mergeDim` ist konsistent, wenn (1) $DW.H_D$ einen Verbund bildet und (2) $\rho^{l \mapsto k}$ für alle Paare $l, k \in DW.H_D$ im Sinne der Definition 5.7 konsistent ist.

5.3.3.2 Operatoren der Fact-Algebra

Diese Gruppe von Operatoren ist dadurch gekennzeichnet, dass sie eine Operation auf genau einer lokalen Faktmenge bzw. auf der globalen Faktmenge repräsentieren. Werden FA-Operatoren auf eine Input-Faktmenge F angewendet, so wird eine Output-Faktmenge F' erzeugt, deren Schema $s(F')$ und/oder Instanzen $i(F')$ modifiziert wurden. Eine wichtige Voraussetzung für die Anwendung der FA-Operatoren ist die Existenz konsolidierter, lokaler Dimensionen, d.h. Dimensionen mit kompatiblen Schemata und Instanzen. Um konsolidierte Dimensionen zu erzeugen, müssen die DA-Operatoren aus Abschnitt 5.3.3.1 angewendet werden. Im Folgenden werden die in der Parser-Komponente implementierten Operatoren der FA in Anlehnung an [BS08] beschrieben. Die zuerst behandelten vier Operatoren σ, π, λ und ζ sind wie in der relationalen Algebra definiert: [KE04]

Select: $\sigma_P(c)$.
Dieser Operator berechnet eine Faktmenge F', welche alle Tupel $t \in F$ enthält, die die Prädikate P erfüllen.

Beispiel 5.25: $\sigma_{kunde='1234010180'}(verkauf)$, angewendet auf die Fakttabelle „mfuB:verkauf“, bewirkt eine Selektion des Fakts mit dem Kunden '1234010180'.

Zugeordnete SQL-MDi-Anweisungen:
`WHERE` Bedingung in `CONVERT MEASURES APPLY`, `CONVERT ATTRIBUTES APPLY`, `RENAME`, `RENAME CONTEXT DIM`, `PREFER`, `AGGREGATE MEASURE`

Project: $\pi_{(L \subset A_C)}(c)$**.**
Dieser Operator reduziert die Dimensionalität von F', indem eine Projektion auf ein oder mehrere dimensionale Attribute L erfolgt. Die Anwendung des π Operators bewirkt eine Zusammenführung von Zellen, da die Koordinaten aufgrund der eingeschränkten Menge von dimensionalen Attributen neu berechnet werden müssen.

Beispiel 5.26: $\pi_{(datum,kunde,mobilnetz,umsatzkategorie)}(verkauf)$, angewendet auf die Fakttabelle „mfuB:verkauf“, bewirkt ein „Ausblenden“ des dimensionalen Attributs „Werbeaktion“.

Zugeordnete SQL-MDi-Anweisung:
`DIM`

Delete measure: $\lambda_{(N \subset M_F)}(c)$**.**
Dieser Operator berechnet eine Faktmenge F', in welcher die Anzahl der Kenngrößen-Attribute durch Projektion auf ein oder mehrere Kenngrößen-Attribute N reduziert wurde.

Beispiel 5.27: $\lambda_{(umsatz\_sonstiges,umsatz\_telefonie)}(verkauf)$, angewendet auf die Fakttabelle „mfuA:verkauf“, bewirkt ein Löschen des Kenngrößen-Attributs „Dauer“.

Zugeordnete SQL-MDi-Anweisung:
`MEASURE`

Rename: ζ**.**
Mit diesem Operator können Namen von Attributen in $s(F')$ umbenannt werden, wobei ζ auf folgende Objekte angewendet werden kann:

- Umbenennen eines dimensionalen Attributs $A \in A_C$: $\zeta_{A' \leftarrow A}(c)$
- Umbenennen eines Kenngrößen-Attributs $M \in M_C$: $\zeta_{M' \leftarrow M}(c)$

Beispiel 5.28: $\zeta_{svnr \leftarrow kunde}(verkauf)$, angewendet auf die Fakttabelle „mfuA:verkauf“, bewirkt ein Umbenennen des dimensionalen Attributes „Kunde“ in „svnr“.

Zugeordnete SQL-MDi-Anweisung:
`DIM` in Verbindung mit dem $->$ Operator (für dimensionale Attribute)
`MEASURE` in Verbindung mit dem $->$ Operator (für Kenngrößen-Attribute)

Convert: $\gamma_{M' \ \theta}(c)$
Mit diesem Operator kann die Domäne eines Kenngrößen-Attributs $M' \in M_C$ geändert werden, indem eine Funktion θ auf $dom(M')$ angewendet wird.

Beispiel 5.29: $\gamma_{umsatz \ usd2Eur()}(verkauf)$, angewendet auf die Fakttabelle „mfuB:verkauf“, bewirkt eine Konvertierung aller Werte des Kenngrößen-Attributs „Umsatz“ von US-Dollar in Euro.

Zugeordnete SQL-MDi-Anweisung:
`CONVERT MEASURES APPLY`

Pivot: $\xi|\chi$.
Mit den beiden Pivot-Varianten ξ und χ können Schema-Instanz-Konflikte (vgl. Abschnitt 3.3.3) aufgelöst werden.

Die Variante ξ (split measure attribute) transformiert einen Teil des Fakt-Kontexts (Instanzen eines dimensionalen (Kontext-)Attributs) in ein oder mehrere Kenngrößen-Attribute: $\xi_{M' \Rightarrow A'}(c)$ erzeugt aus einem Quell-Kenngrößen-Attribut M' und einem Quell-Kontextattribut A' für jeden Wert $v \in A'$ ein neues Kenngrößen-Attribut mit den entsprechenden Werten aus M'. Aufgrund der Einführung zusätzlicher Kenngrößen-Attribute, wird die Information in F' komprimiert, d.h. Tupel werden zusammengeführt.

Beispiel 5.30: $\xi_{umsatz \Rightarrow umsatzkategorie}(verkauf)$ bewirkt, angewendet auf die Fakttabelle „mfuB:verkauf“, dass für jeden Wert des dimensionalen Attributs „Umsatzkategorie“, nämlich „Umsatz_Telefonie“ und „Umsatz_Sonstiges“, ein neues Kenngrößen-Attribut mit den entsprechenden Umsatz-Werten eingeführt wird; das dimensionale Attribut „Umsatzkategorie“ und das Kenngrößen-Attribut „Umsatz“ werden entfernt. Nach Anwendung des ξ Operators entspricht das Schema der Fakttabelle „mfuB:verkauf“ jenem der Fakttabelle „mfuA:verkauf“.

Die Variante χ (merge measure attributes) führt mehrere Kenngrößen-Attribute in ein einziges zusammen, wobei der Kontext in einem zusätzlichen dimensionalen Attribut festgehalten wird: $\chi_{M_L \Rightarrow M_x, A_x}(c)$ erzeugt aus einer Menge von Quell-Kenngrößen-Attributen $M_L \subseteq M_C = \{M_1, ..., M_k\}$ ein neues, einzelnes Kenngrößen-Attribut M_x, für dessen Werte der Kontext in einem neuen, dimensionalen (Kontext-)Attribut A_x gespeichert wird. Im Gegensatz zur Variante ξ wird bei dieser Variante die Information in F' dekomprimiert, d.h. Tupel werden geteilt.

Beispiel 5.31: $\chi_{\{umsatz\_telefonie, umsatz\_sonstiges\} \Rightarrow umsatz, umsatzkategorie}(verkauf)$ bewirkt, angewendet auf die Fakttabelle „mfuA:verkauf“, dass für alle Umsätze ein neues Kenngrößen-Attribut „Umsatz“ sowie ein neues dimensionales Attribut „Umsatzkategorie“ mit den Werten „Umsatz_Telefonie“ und „Umsatz_Sonstiges“ eingeführt wird; die Quell-Kenngrößen-Attribute „Umsatz_Telefonie“ und „Umsatz_Sonstiges“ werden entfernt. Nach Anwendung des χ Operators entspricht das Schema der Fakttabelle „mfuA:verkauf“ jenem der Fakttabelle „mfuB:verkauf“.

Zugeordnete SQL-MDi-Anweisungen:
`PIVOT MEASURE` (für ξ)
`PIVOT MEASURES` (für χ)

Enrich dimensions: $\epsilon_{A'=v}(c)$.
Dieser Operator erhöht die Dimensionalität von F', indem ein zusätzliches dimensionales Attribut A' mit v als fixem Wert eingeführt wird. ϵ kann beispielsweise verwendet werden um den einzelnen Fakten eine Information über das Quell-DW zuzuordnen.

Beispiel 5.32: $\epsilon_{source='dw1'}$, angewendet auf die Fakttabelle „mfuA:verkauf“, bewirkt die Einführung eines neuen dimensionalen Attributs „source“ mit dem Wert 'dw1'.

Zugeordnete SQL-MDi-Anweisung:
`TRACKING SOURCE AS DIMENSION`

Merge facts: μ.
Mit diesem Operator kann die semantische Beziehung zwischen Fakten mit überlappenden Koordinaten spezifiziert werden. Seien F_{C1}, F_{C2} zwei Faktmengen mit den Schemata $G = S_{C1}$, $H = S_{C2}$ mit $G \cap H = \{A_1, ..., A_n, M_1, ...M_M\}$ und $Op = \{prefer, sum, avg, min, max\}$ eine vordefinierte Menge von Operatoren. Weiters, sei $N \subseteq \{M_1, ..., M_m\}$ eine Teilmenge der Kenngrößen-Attribute beider Faktmengen, für welche ein Operator $\theta \in Op$ angewendet werden soll. $\mu(g[N, \theta]h)$ berechnet als Ergebnis eine Faktmenge F' mit dem Schema $S' = G \cup H$ und der Instanz $c'(S') = \{t(S') | \exists t_g \in G, \exists t_h \in H : t(G \backslash N) = t_g(G \backslash N), t(H \backslash N) = t_h(H \backslash N), t(N) = t_g(N) \theta t_h(N)\}$.

Der *prefer* Operator eignet sich für überlappende Fakten, die in einer Identitätsbeziehung zueinander stehen; die anderen (Aggregations-)Operatoren werden für in einer Kontextbeziehung zueinander stehende, überlappende Fakten verwendet (vgl. Abschnitt 3.3.3). Der μ Operator wird nur im globalen Kontext, d.h. auf einer globalen Faktmenge, angewendet.

Beispiel 5.33: $\mu(g(umsatz, SUM)h)$, mit $g \subset c(G)$ und $h \subset c(H)$, angewendet für die Instanziierung der globalen Fakttabelle, bewirkt eine Summierung der lokalen Kenngrößen des Attributs „Umsatz“ in eine globale Kenngröße eines Attributs „Umsatz“, für überlappende Fakten g und h.

Zugeordnete SQL-MDi-Anweisungen:
`PREFER`, `AGGREGATE MEASURE`

Auf Basis der in diesem Abschnitt erläuterten DA/FA-Operatoren soll eine Operator-Baumstruktur erzeugt werden. Der folgende Abschnitt veranschaulicht die Komposition der Operatoren zu einer solchen Baumstruktur.

5.3.4 Komposition der DA/FA-Operatoren

Die Operator-Baumstruktur legt fest, in welcher Reihenfolge die DA/FA-Operatoren durch eine Prozessor-Komponente ausgewertet werden müssen. Das heißt, dass die Komponente den Baum durch Anwendung eines spezifizierten Traversierungs-Algorithmus [Ski98], ohne Umstrukturierungen vorzunehmen, vollständig verarbeiten kann. Im folgenden Abschnitt wird zunächst die logische Struktur des Baums erläutert, entsprechend der die Parser-Komponente die DA/FA-Operatoren anordnet, bevor die der Struktur zugrunde liegenden Annahmen beschrieben werden.

5.3.4.1 Logische Struktur des Operator-Baums

Der Aufbau der Operator-Baumstruktur lässt sich aus der Zuordenbarkeit der einzelnen Operatoren zu lokalen Dimensionen, lokalen Faktmengen (lokalen Würfeln) bzw. zur globalen Faktmenge (zum globalen Würfel) ableiten. Um eine Baumstruktur auf Basis dieser Operator-Zuordnung konstruieren zu können, werden „Strukturknoten“ für lokale und globale Dimensionen bzw. für lokale und globale Würfel eingeführt. Um die Zusammenführung von lokalen

Dimensionen zu einer globalen Dimension bzw. von lokalen Würfeln zu einem globalen Würfel darstellen zu können, werden zwei weitere Strukturknoten für die aus SQL-MDi bekannten Anweisungen `MERGE DIMENSIONS` und `MERGE CUBES` verwendet. Diese Knoten repräsentieren die von der Prozessor-Komponente zu implementierenden `merge`-Algorithmen. Der Aufbau der DA/FA-Baumstruktur lässt sich daher wie folgt beschreiben.

- Die Wurzel des Baums root wird durch einen „dummy“-Knoten ohne konkrete Funktion repräsentiert.
- Die `mergeDim`-Algorithmen werden als Kindknoten md von root modelliert.
  - Ein md-Knoten enthält eine Menge von Knoten für lokale Dimensionen d sowie einen Knoten für die globale Dimension gd als Kindknoten.
  - Ein d-Knoten enthält jene DA-Operatoren als Kindknoten, welche auf dieser Dimension ausgeführt werden sollen.
  - Ein gd-Knoten enthält in der aktuellen SQL-MDi-Version keine Kindknoten.
- Der `mergeFacts`-Algorithmus wird als Kindknoten mc von root modelliert.
  - Ein mc-Knoten enthält eine Menge von Knoten für lokale Würfel c sowie einen Knoten für den globalen Würfel gc als Kindknoten.
  - Ein c-Knoten enthält jene FA-Operatoren als Kindknoten, welche auf diesem Würfel ausgeführt werden sollen.
  - Ein gc-Knoten enhält jene FA-Operatoren als Kindknoten, welche auf diesem globalen Würfel ausgeführt werden sollen.

Abbildung 5.2 zeigt einen Ausschnitt aus der aus Listing 5.19 generierbaren Baumstruktur:

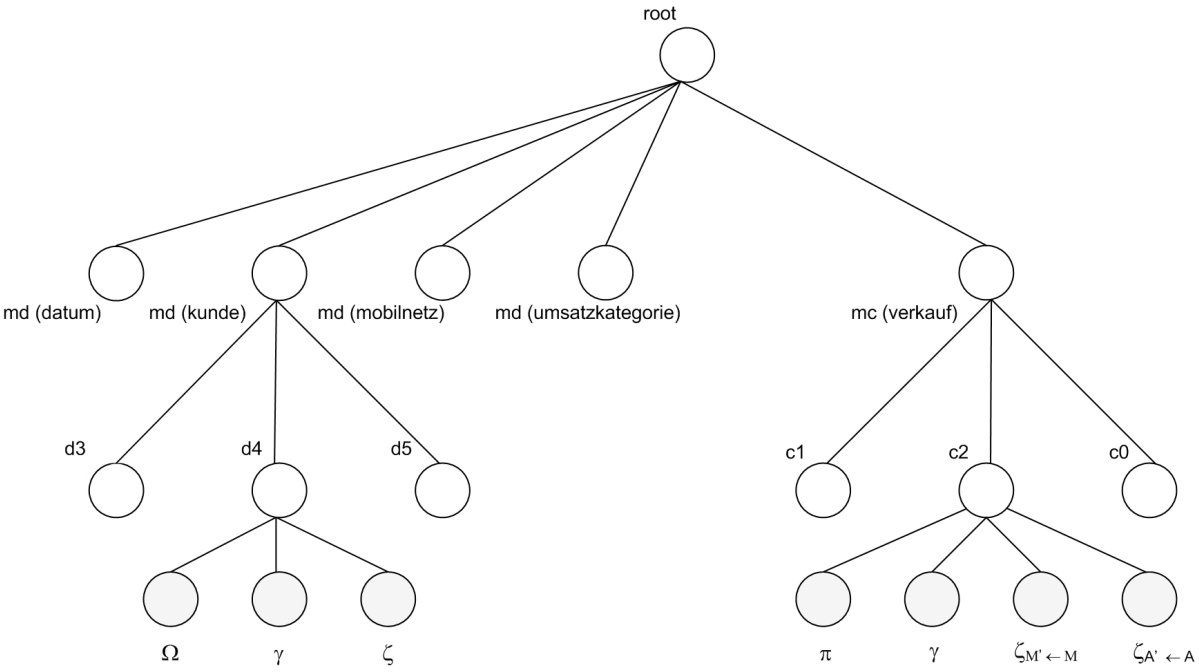


Abbildung 5.2: Operator-Baumstruktur für Listing 5.19

Die Operator-Baumstruktur der Abbildung 5.2 enthält vier md-Knoten und einen mc-Knoten, entsprechend der `MERGE DIMENSIONS` und `MERGE CUBES` Anweisungen des Listing 5.19. Die Abbildung zeigt eine weitere Präzisierung des md-Knotens für die Dimension „Kunde“, welche die zwei lokalen Dimensionen d3 und d4 und die globale Dimension d5 als Kindknoten umfasst. Für die Dimension d4 sind die zugeordneten DA-Operatoren abgebildet, nämlich Ω, γ und ζ. Außerdem veranschaulicht die Abbildung die Struktur des mc-Knoten, welcher die zwei lokalen Würfel c1 und c2 sowie den globalen Würfel c0 als Kindknoten enthält. Für den Würfel c2 sind die zugeordneten FA-Operatoren abgebildet, nämlich π, γ, $\zeta_{M' \leftarrow M}$ und $\zeta_{A' \leftarrow A}$.

Die DA/FA-Operatoren sind so in der Baumstruktur angeordnet, dass eine Prozessor-Komponente diese, ohne Umstrukturierungen vorzunehmen, vollständig verarbeiten kann. Die Determinanten für die definierte Anordnung der DA/FA-Operatoren in der Baumstruktur, welche die Auswertungsreihenfolge bestimmt, werden im nächsten Abschnitt erläutert.

5.3.4.2 Auswertungsreihenfolge der DA/FA-Operatoren

Für die Prozessor-Komponente wird als Traversierungsstrategie „Depth-First-Search“ [Ski98] zur Verarbeitung der Operator-Baumstruktur angenommen. Betrachtet man den Baum der Abbildung 5.2, werden zuerst die der ersten lokalen Dimension (d1) des Knotens „md (datum)“ zugeordneten DA-Operatoren „von links nach rechts“ auswertet, anschließend die der zweiten lokalen Dimension (d2), anschließend die der globalen Dimension (d0), anschließend die der ersten lokalen Dimension (d3) des Knotens „md (kunde)“, ..., bis zuletzt die FA-Operatoren des globalen Würfels (c0) verarbeitet werden.

Die Anordnung der DA/FA-Operatoren wird durch zwei Aspekte bestimmt:

- Operatoren, welche die Ergebnismenge verkleinern, sollen zuerst angewendet werden, wodurch eine Optimierung der Abfrage auf logischer Ebene erreicht wird.

  Analysiert man die Operatoren hinsichtlich dieser Eigenschaft, kommt man zum Ergebnis, dass bei den FA-Operatoren σ, π, ξ und λ bzw. bei den DA-Operatoren α die Ergebnismenge verkleinern. Für die FA-Operatoren σ, π, ξ und λ gilt: σ schränkt die Faktmenge durch Selektion ein; π reduziert die Anzahl der dimensionalen Attribute einer Fakttabelle durch Projektion; ξ bewirkt eine Zusammenführung von Tupeln der Fakttabelle, wodurch die Faktmenge verkleinert wird, und λ reduziert die Anzahl der Kenngrößen-Attribute einer Fakttabelle durch Projektion. Für den DA-Operator α gilt: α reduziert die Anzahl der Dimensionsinstanzen, indem eine Klassifikationsstufe einer Dimension entfernt wird.

  Diese Operatoren werden daher als äußerst linke Kindknoten eines Würfel- bzw. Dimensions-Knotens eingefügt, damit sie von der Prozessor-Komponente zuerst abgearbeitet werden.

- Operatoren, welche nicht unabhängig von allen anderen Operatoren ausgeführt werden können, sollen so in die Operator-Baumstruktur eingefügt werden, dass eine vollständige Verarbeitung derselben gewährleistet ist.

Das heißt, dass alle Operatoren der Baumstruktur angewendet werden und ein Ergebnis berechnen müssen.

Beispiel 5.34: Wird ein Kenngrößen-Attribut mit $\zeta_{M' \leftarrow M}$ umbenannt, kann kein $\gamma_{M\ \theta}(c)$ Operator zur Konvertierung der Domäne des Kenngrößen-Attributs angewendet werden, da dieser Operator den alten Namen des Kenngrößen-Attributs verwendet. Eine Verarbeitung beider Operatoren wäre nur möglich, wenn γ vor ζ ausgewertet wird.

Diese beiden Operatoren können somit nicht unabhängig voneinander in die Operator-Baumstruktur eingefügt werden.

Für einen Operator, welcher nicht unabhängig von allen anderen Operatoren angewendet werden kann, ist vor dem Einfügen in den Baum zu prüfen, ob eine Abhängigkeit zu einem bereits eingefügten Operator besteht.

Einfüge-Regeln für nicht-unabhängige Operatoren

Im Folgenden wird für die nicht-unabhängig anwendbaren Operatoren die jeweilige Abhängigkeit erläutert, sowie eine entsprechende Einfüge-Regel definiert. Zusammengehörige Abhängigkeiten und Regeln können durch eine identische Nummerierung identifiziert werden (z.B. Abhängigkeit 1.1 wird mit der Regel 1.1 behandelt).

1. **Rename:** $\zeta_{M' \leftarrow M}(c)$ **(FA).**

   Abhängigkeit 1.1:
   Für diesen Operator ergibt sich eine Abhängigkeit aufgrund der getroffenen Annahme, dass alle Operatoren, welche ein oder mehrere Kenngrößen-Attribute als Parameter enthalten, die ursprünglichen Namen der Kenngrößen-Attribute verwenden. Aus diesem Grund wird $\zeta_{M' \leftarrow M}(c)$ erst am Ende verarbeitet, da ansonsten andere abhängige Operatoren nicht mehr korrekt angewendet werden können.

   Regel 1.1:
   $\zeta_{M' \leftarrow M}(c)$ wird als äußerst rechter Kindknoten eines Würfel-Knotens c eingefügt.

   Abhängigkeit 1.2:
   Eine zweite Abhängigkeit bewirkt eine Ausnahme von Regel 1.1: $\zeta_{M' \leftarrow M}(c)$ beeinflusst einen $\chi_{M_L \Rightarrow M_x, A_x}(c)$ Operator, wenn $M \in M_L \subseteq M_C = \{M_1, ..., M_k\}$, da die Namen der Kenngrößen-Attribute in M_L zu Instanzen eines dimensionalen (Kontext-)Attributs werden. Es besteht daher die Möglichkeit durch Anwendung von $\zeta_{M' \leftarrow M}(c)$ die zu erzeugenden Instanzen zu beeinflussen.

   Regel 1.2:
   $\zeta_{M' \leftarrow M}(c)$ wird links vor einem $\chi_{M_L \Rightarrow M_x, A_x}(c)$ Operator als Kindknoten eines Würfel-Knotens c eingefügt, wenn $M \in M_L \subseteq M_C = \{M_1, ..., M_k\}$.

2. **Rename:** $\zeta_{A' \leftarrow A}(c)$ **(FA).**

   Abhängigkeit 2.1:
   Für diesen Operator ergibt sich eine Abhängigkeit aufgrund der getroffenden Annahme, dass alle Operatoren, welche ein oder mehrere dimensionale Attribute als Parameter

enthalten, die ursprünglichen Namen der dimensionalen Attribute verwenden. Aus diesem Grund wird $\zeta_{A' \leftarrow A}(c)$ erst am Ende verarbeitet, da ansonsten andere abhängige Operatoren nicht mehr korrekt angewendet werden können.

Regel 2.1:
$\zeta_{A' \leftarrow A}(c)$ wird als äußerst rechter Kindknoten eines Würfel-Knotens c eingefügt.

3. **Rename:** $\zeta_{l' \leftarrow l}(d)$ **bzw.** $\zeta_{a' \leftarrow l.a}(d)$ **(DA).**

   Abhängigkeit 3.1:
   Für diesen Operator ergibt sich eine Abhängigkeit aufgrund der getroffenden Annahme, dass alle Operatoren, welche ein oder mehrere Klassifikationsstufen bzw. nichtdimensionale Attribute als Parameter enthalten, die ursprünglichen Namen verwenden. Aus diesem Grund werden $\zeta_{l' \leftarrow l}(d)$ bzw. $\zeta_{a' \leftarrow l.a}(d)$ erst am Ende verarbeitet, da ansonsten andere abhängige Operatoren nicht mehr korrekt angewendet werden können.

   Regel 3.1:
   $\zeta_{l' \leftarrow l}(d)$ bzw. $\zeta_{a' \leftarrow l.a}(d)$ werden als äußerst rechte Kindknoten eines Dimensions-Knotens d eingefügt.

4. **Change:** $\delta_{w \leftarrow v(l.k_j)}(d)$ **(DA).**

   Abhängigkeit 4.1:
   Dieser Operator besitzt eine Abhängigkeit zu einem $\xi_{M' \Rightarrow A'}(c)$ Operator, wenn $f(A') \subseteq members(l)$, da eine Änderung der Werte des Kontextattributs geänderte Kenngrößen-Attribut-Namen (nach Ausführung des ξ Operators) bewirkt. Für diese Abhängigkeit ist eine Einfüge-Regel erforderlich, da $\delta_{w \leftarrow v(l.k_j)}(d)$ zur Festlegung der Kenngrößen-Attribut-Namen für das „Pivoting“ (mit der `RENAME CONTEXT DIM` Anweisung) verwendet werden kann.

   Regel 4.1:
   Ein $\delta_{w \leftarrow v(l.k_j)}(d)$, für die `RENAME CONTEXT DIM` Anweisung, wird links vor einem $\xi_{M' \Rightarrow A'}(c)$ Operator als Kindknoten eines Würfel-Knotens c eingefügt, wenn $f(A') \subseteq members(l)$.

   Abhängigkeit 4.2:
   Dieser Operator besitzt eine Abhängigkeit zu einem $\Omega_{m \mapsto v}(d)$ Operator, wenn $l = level(m)$, d.h. werden Instanzen der zu überschreibenden Klassifikationsstufe $level(m)$, $T_i = members(level(m))$, geändert, so muss $\delta_{w \leftarrow v(l.k_j)}(d)$ vor Ω ausgewertet werden.

   Regel 4.2:
   $\delta_{w \leftarrow v(l.k_j)}(d)$ wird links vor einem $\Omega_{m \mapsto v}(d)$ Operator als Kindknoten eines Dimensions-Knotens d eingefügt, wenn $l = level(m)$.

5. **Convert:** $\gamma_{(l.k_j)\theta}(d)$ **(DA).**

   Abhängigkeit 5.1:
   Dieser Operator besitzt eine Abhängigkeit zu einem $\Omega_{m \mapsto v}(d)$ Operator, wenn $l = level(m)$, d.h. werden Instanzen der zu überschreibenden Klassifikationsstufe $level(m)$, $T_i = members(level(m))$, konvertiert, so muss $\gamma_{(l.k_j)\theta}(d)$ vor Ω ausgewertet werden.

   Regel 5.1:
   $\gamma_{(l.k_j)\theta}(d)$ wird links vor einem $\Omega_{m \mapsto v}(d)$ Operator als Kindknoten eines Dimensions-Knotens d eingefügt, wenn $l = level(m)$.

6. **Override rollup:** $\Omega_{m \mapsto v}(d)$ **(DA).**

   Dieser Operator hat Abhängigkeiten zu $\delta_{w \leftarrow v(l.k_j)}(d)$ und $\gamma_{(l.k_j)\theta}(d)$. Es sind daher die Regeln 4.2 und 5.1 aus Sicht des Ω Operators anzuwenden.

7. **Convert:** $\gamma_{M'\ \theta}(c)$ **(FA).**

   Abhängigkeit 7.1:
   Dieser Operator besitzt eine Abhängigkeit zu einem $\xi_{M' \Rightarrow A'}(c)$ Operator, wenn $M'(\gamma) = M'(\xi)$, da die Anwendung dieser Pivot-Variante geänderte Kenngrößen-Attribut-Namen bewirkt und $\gamma_{M'\ \theta}(c)$ somit nicht mehr korrekt verarbeitet werden kann.

   Regel 7.1:
   $\gamma_{M'\ \theta}(c)$ wird links vor einem $\xi_{M' \Rightarrow A'}(c)$ Operator als Kindknoten eines Würfel-Knotens c eingefügt, wenn $M'(\gamma) = M'(\xi)$.

   Abhängigkeit 7.2:
   Für diesen Operator besteht eine Abhängigkeit zu einem $\chi_{M_L \Rightarrow M_x, A_x}(c)$ Operator, wenn $M' \in M_L$, da die Anwendung dieser Pivot-Variante geänderte Kenngrößen-Attribut-Namen bewirkt und $\gamma_{M'\ \theta}(c)$ somit nicht mehr korrekt verarbeitet werden kann.

   Regel 7.2:
   $\gamma_{M'\ \theta}(c)$ wird links vor einem $\chi_{M_L \Rightarrow M_x, A_x}(c)$ Operator als Kindknoten eines Würfel-Knotens c eingefügt, wenn $M' \in M_L$.

   Abhängigkeit 7.3:
   Trifft Abhängigkeit 1.2 zu, so kann eine zusätzliche Abhängigkeit zu einem vor den $\chi_{M_L \Rightarrow M_x, A_x}(c)$ Operator platzierten $\zeta_{M' \leftarrow M}(c)$ Operator bestehen (Regel 1.2), wenn $M'(\gamma) = M$.

   Regel 7.3:
   $\gamma_{M'\ \theta}(c)$ wird links vor einem $\zeta_{M' \leftarrow M}(c)$ Operator als Kindknoten eines Würfel-Knotens c eingefügt, wenn Regel 1.2 angewendet wurde und $M'(\gamma) = M$.

8. **Pivot (Split measure attributes):** $\xi_{M' \Rightarrow A'}(c)$ **(FA).**

   Dieser Operator hat Abhängigkeiten zu $\delta_{w \leftarrow v(l.k_j)}(d)$ und $\gamma_{M'\ \theta}(c)$ Operatoren. Es sind daher die Regeln 4.1 und 7.1 aus Sicht des ξ Operators anzuwenden.

9. **Pivot (Merge measure attributes:** $\chi_{M_L \Rightarrow M_x, A_x}(c)$ **(FA).**

   Dieser Operator hat Abhängigkeiten zu $\zeta_{M' \leftarrow M}(c)$ und $\gamma_{M'\ \theta}(c)$ Operatoren. Es sind daher die Regeln 1.2, 7.2 und 7.3 aus Sicht des χ Operators anzuwenden.

Trotz der Anwendung der Einfüge-Regeln, besteht die Möglichkeit, dass eine einer SQL-MDi-Abfrage entsprechende Baumstruktur, Kombinationen von DA/FA-Operatoren enthält, welche in einem logischen Konflikt stehen. Diese Konflikte im „Abfrage-Kontext“ werden im nächsten Abschnitt behandelt.

5.3.4.3 Konflikte im Abfrage-Kontext

Die Transformation einer syntaktisch korrekten SQL-MDi-Abfrage in eine Operator-Baumstruktur kann logische Konflikte zwischen DA/FA-Operatoren hervorrufen. So bewirkt beispielsweise ein $\lambda_{(N \subset M_F)}(c)$ Operator und ein $\gamma_{M'\ \theta}(c)$ Operator, wobei $M' \notin N$, einen ungültigen Abfragekontext, da die Werte eines gelöschten Kenngrößen-Attributs nicht mehr konvertiert werden können; gilt hingegen $M' \in N$, so besteht klarerweise kein Konflikt. Die Graph-Darstellung der Abbildung 5.3 zeigt zwischen Operatoren mögliche Kontext-Konflikte, welche im Zuge der Transformation einer SQL-MDi-Abfrage durch die Parser-Komponente auftreten können.

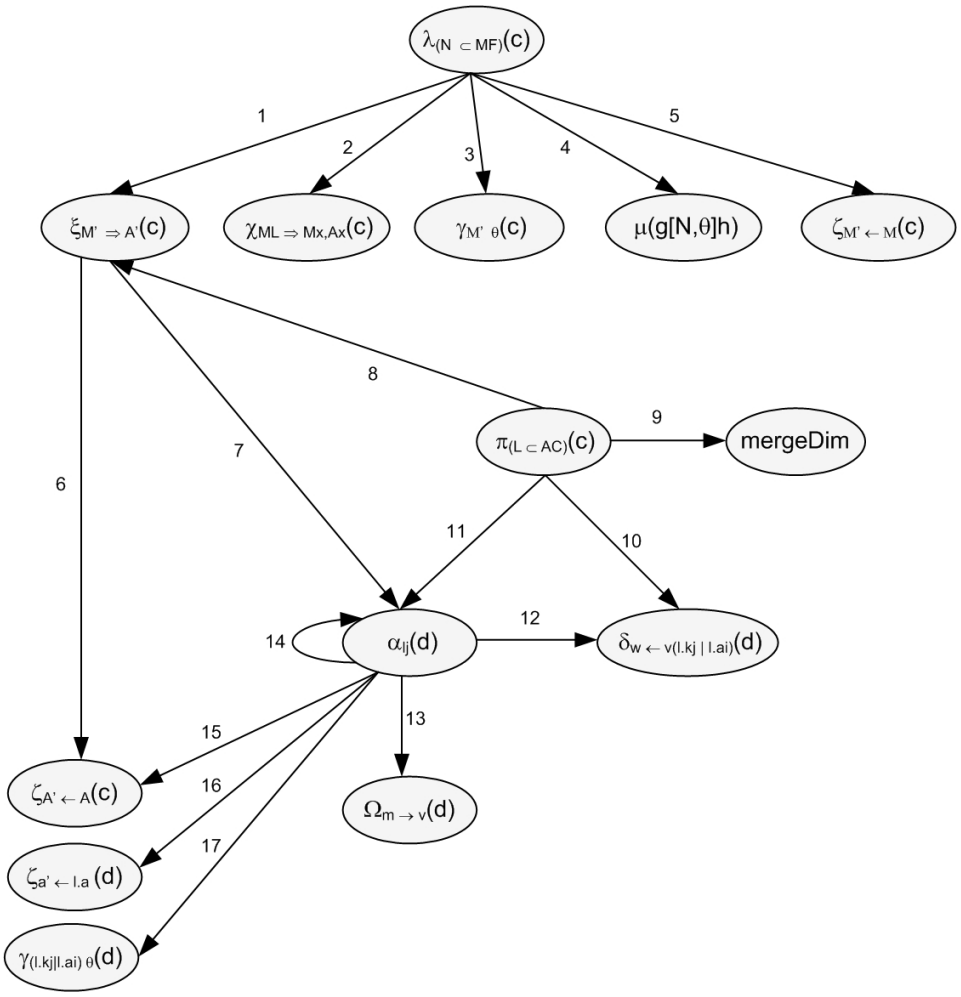


Abbildung 5.3: Mögliche Konflikte zwischen DA/FA-Operatoren

Ein nummerierter Pfeil zwischen zwei Operatoren a und b in der Abbildung 5.3 symbolisiert einen logischen Konflikt, unter der Annahme, dass a und b demselben Strukturknoten, d.h. demselben Dimensions- oder Würfel-Knoten, zugeordnet sind. Allgemein ausgedrückt, besagt ein logischer Konflikt, a → b, dass die Verarbeitung von b nicht möglich ist wenn a bereits

verarbeitet wurde. Allen Konflikten gemeinsam ist, dass ein vom Operator b verwendeter Parameter durch den Operator a entfernt bzw. ausgeblendet wurde. Im Folgenden werden die durch die nummerierten Pfeile dargestellten logischen Konflikte erläutert:

1. Zwischen $\lambda_{(N \subset M_F)}(c)$ und $\xi_{M' \Rightarrow A'}(c)$ besteht ein logischer Konflikt, wenn $M' \notin N$: Ein gelöschtes Kenngrößen-Attribut kann nicht für das „Pivoting“ verwendet werden.
2. Zwischen $\lambda_{(N \subset M_F)}(c)$ und $\chi_{M_L \Rightarrow M_x, A_x}(c)$ besteht ein logischer Konflikt, wenn $\exists M \in M_L : M \notin N$: Ein gelöschtes Kenngrößen-Attribut kann nicht für das „Pivoting“ verwendet werden.
3. Zwischen $\lambda_{(N \subset M_F)}(c)$ und $\gamma_{M'\ \theta}(c)$ besteht ein logischer Konflikt wenn $M' \notin N$: Ein gelöschtes Kenngrößen-Attribut kann nicht konvertiert werden.
4. Zwischen $\lambda_{(N \subset M_F)}(c)$ und $\mu(g[N, \theta]h)$ besteht ein logischer Konflikt, wenn $\exists M \in N(\mu) : M \notin N(\lambda)$: Ein gelöschtes Kenngrößen-Attribut kann nicht bei der Behandlung überlappender Fakten berücksichtigt werden.
5. Zwischen $\lambda_{(N \subset M_F)}(c)$ und $\zeta_{M' \leftarrow M}(c)$ besteht ein logischer Konflikt, wenn $M \notin N$: Ein gelöschtes Kenngrößen-Attribut kann nicht umbenannt werden.
6. Zwischen $\xi_{M' \Rightarrow A'}(c)$ und $\zeta_{A' \leftarrow A}(c)$ besteht ein logischer Konflikt, wenn $A = A'(\xi)$: Ein durch das „Pivoting“ mit dem ξ Operator entferntes dimensionales Attribut kann nicht umbenannt werden.
7. Zwischen $\xi_{M' \Rightarrow A'}(c)$ und $\alpha_{l_j}(d)$ besteht ein logischer Konflikt, wenn $f(A') \subseteq members(l_j)$: Die mit einem durch das „Pivoting“ mit dem ξ Operator entfernten dimensionalen Attribut verbundene Klassifikationsstufe kann nicht gelöscht werden.
8. Zwischen $\xi_{M' \Rightarrow A'}(c)$ und $\pi_{(L \subset A_C)}(c)$ besteht ein logischer Konflikt, wenn $A' \notin L$: Ein mit einer Projektion ausgeblendetes dimensionales Attribut kann nicht für das „Pivoting“ mit dem ξ Operator verwendet werden.
9. Zwischen $\pi_{(L \subset A_C)}(c)$ und dem `mergeDim`-Algorithmus, wenn $\exists D \in mergeDim, \exists A_i \notin L$, wobei A_i mit der Dimension D verbunden ist: Die mit einem dimensionalen Attribut, welches mit einer Projektion ausgeblendet wurde, verbundene Dimension kann nicht mit dem `mergeDim`-Algorithmus zusammengeführt werden.
10. Zwischen $\pi_{(L \subset A_C)}(c)$ und $\delta_{w \leftarrow v(l.k_j | l.a_i)}(d)$ besteht ein logischer Konflikt, wenn $\exists A_i \notin L : f(A_i) \subseteq members(l)$: Die Instanzen eines mit einer Projektion ausgeblendeten dimensionalen Attributs können nicht geändert werden.
11. Zwischen $\pi_{(L \subset A_C)}(c)$ und $\alpha_{l_j}(d)$ besteht ein logischer Konflikt, wenn $\exists A_i \notin L$ und A_i mit der Dimension d verknüpft ist: Es kann keine Klassifikationsstufe einer Dimension gelöscht werden, wenn die Dimension durch Ausblenden des verbundenen dimensionalen Attributs ebenfalls ausgeblendet wurde.
12. Zwischen $\alpha_{l_j}(d)$ und $\delta_{w \leftarrow v(l.k_j | l.a_i)}(d)$ besteht ein logischer Konflikt, wenn $l_j = l$: Instanzen eines dimensionalen bzw. nicht-dimensionalen Attributs einer gelöschten Klassifikationsstufe können nicht geändert werden.
13. Zwischen $\alpha_{l_j}(d)$ und $\Omega_{m \mapsto v}(d)$ besteht ein logischer Konflikt, wenn $l = level(m)$: Die roll-up Beziehung für eine gelöschte Klassifikationsstufe kann nicht verändert werden.
14. Zwischen $\alpha_{l_j}(d)$ und $\alpha_{l_j}(d)$ besteht ein logischer Konflikt, wenn $l_j = l_j$: Eine gelöschte Klassifikationsstufe kann nicht gelöscht werden.

15. Zwischen $\alpha_{l_j}(d)$ und $\zeta_{A' \leftarrow A}(c)$ besteht ein logischer Konflikt, wenn $f(A) \subseteq members(l_j)$: Ein dimensionales Attribut kann nicht umbenannt werden, wenn die damit verbundene Klassifikationsstufe einer Dimension gelöscht wurde.
16. Zwischen $\alpha_{l_j}(d)$ und $\zeta_{a' \leftarrow l.a}(d)$ besteht ein logischer Konflikt, wenn $l_j = l$: Ein Attribut einer gelöschten Klassifikationsstufe kann nicht umbenannt werden.
17. Zwischen $\alpha_{l_j}(d)$ und $\gamma_{(l.k_j|l.a_i)\theta}(d)$ besteht ein logischer Konflikt, wenn $l_j = l$: Ein Attribut einer gelöschten Klassifikationsstufe kann nicht konvertiert werden.

5.4 SQL-MDi-Anweisungen vs. DA/FA-Operatoren

Die Tabellen 5.1 und 5.2 dieses Abschnitts stellen die im Abschnitt 5.2 präsentierten SQL-MDi-Anweisungen zur Auflösung von semantischen Heterogenitäten auf Schema- und Instanzebene (vgl. Abschnitt 3.3.3) den im Abschnitt 5.3.3 vorgestellten Operatoren der DA/FA gegenüber.

SQL-MDi-Anweisung	**DA/FA-Operator**
Konflikte auf Schemaebene	
Dimensionalität	
DEFINE CUBE mfuB::verkauf AS c2 (MEASURE c2.gespraechsdauer, MEASURE c2.umsatz, DIM c2.datum, DIM c2.kunde, DIM c2.mobilnetz, DIM c2.umsatzkategorie)	$\pi_{(datum,kunde,mobilnetz,umsatzkategorie)}(verkauf)$
Unterschiedliche Aggregationshierarchien	
DEFINE CUBE mfuB::verkauf AS c2 (MEASURE ..., DIM ..., DIM c2.datum (MAP LEVELS c2.datum([datum],[monat],[jahr])))	$\alpha_{quartal}(datum)$
Namenskonflikte	
DEFINE CUBE mfuB::verkauf AS c2 (MEASURE c2.gespraechsdauer −> dauer, ...)	$\zeta_{dauer \leftarrow gespraechsdauer}(verkauf)$
DEFINE CUBE mfuB::verkauf AS c2 (...,DIM c2.kunde −> svnr, ...)	$\zeta_{svnr \leftarrow kunde}(verkauf)$
DEFINE CUBE mfuB::verkauf AS c2 (..., DIM c2.datum(MAP LEVELS c2.datum([datum], [month −> monat], [jahr])),...)	$\zeta_{monat \leftarrow month}(datum)$
MERGE DIMENSIONS c1.kunde AS d1, c2.kunde AS d2 INTO c0.kunde AS d0 (MATCH ATTRIBUTES d1.bezeichnung IS d2.name)	$\zeta_{bezeichnung \leftarrow name}(datum)$
Domänenkonflikte	
DEFINE CUBE mfuB::verkauf AS c2 ... (CONVERT MEASURES APPLY usd2Eur() FOR c2.umsatz DEFAULT)	$\gamma_{umsatz\ usd2Eur()}(verkauf)$
MERGE DIMENSIONS c1.kunde AS d1, c2.kunde AS d2 INTO c0.kunde AS d0 (CONVERT ATTRIBUTES APPLY usd2Eur() FOR d2.grundgebuehr DEFAULT)	$\gamma_{tarif.grundgebuehr\ usd2Eur()}(kunde)$
DEFINE CUBE mfuA::verkauf AS c1 ... (ROLLUP c1.datum_std TO LEVEL c1.datum[datum])	$\alpha_{datum\_std}(datum)$

Tabelle 5.1: SQL-MDi-Anweisungen vs. DA/FA-Operatoren 1/2

SQL-MDi-Anweisung	DA/FA-Operator
Schema-Instanz-Konflikte	
DEFINE CUBE mfuA::verkauf AS c1 (MEASURE c1.umsatz_telefonie, MEASURE c1.umsatz_sonstiges, DIM c1.datum, DIM c1.svnr −> kunde, DIM c1.mobilnetz, PIVOT MEASURES c1.umsatz_telefonie, c1.umsatz_sonstiges INTO c1.umsatz USING c1.umsatzkategorie)	$\chi_{\{umsatz\_telefonie,umsatz\_sonstiges\} \Rightarrow umsatz,umsatzkategorie}(verkauf)$ $\lambda_{(umsatz\_sonstiges,umsatz\_telefonie)}(verkauf)$
DEFINE CUBE mfuA::verkauf AS c2 (MEASURE c2.umsatz, DIM c2.datum, DIM c2.svnr, DIM c2.mobilnetz, DIM c2.umsatzkategorie, PIVOT MEASURE c2.umsatz BASED ON c2.umsatzkategorie)	$\xi_{umsatz \Rightarrow umsatzkategorie}(verkauf)$ $\lambda_{(umsatz)}(verkauf)$
Konflikte auf Instanzebene	
Überlappende/Disjunkte Fakten	
MERGE CUBES c1, c2 INTO c0 ON datum, kunde, mobilnetz, umsatzkategorie (AGGREGATE MEASURE umsatz IS SUM OF umsatz DEFAULT)	$\mu(g(umsatz, SUM)h)$
MERGE CUBES c1, c2 INTO c0 ON datum, kunde, mobilnetz, umsatzkategorie (TRACKING SOURCE AS DIMENSION source(VARCHAR(10)) IS 'mfuA' WHERE SOURCE()='c1', IS 'mfuB' WHERE SOURCE()='c2')	$\epsilon_{source='dw1'}$, $\epsilon_{source='dw2'}$
MERGE CUBES c1, c2 INTO c0 ON datum, kunde, mobilnetz, umsatzkategorie (PREFER c1.umsatz DEFAULT)	$\mu(g(umsatz, PREFER)h)$
Namenskonflikte	
MERGE DIMENSIONS c1.mobilnetz AS d1, c2.mobilnetz AS d2 INTO c0.mobilnetz AS d3 (RENAME d1.mobilnetz >> 'HandyTelInc' WHERE c1.mobilnetz.mobilnetz='HandyTel')	$\delta_{'HandyTelInc' \leftarrow 'HandyTel'(mobilnetz)}(mobilnetz)$
Überlappende/Disjunkte Dimensionsinstanzen	
MERGE DIMENSIONS c1.mobilnetz AS d1, c2.mobilnetz AS d2 INTO c0.mobilnetz AS d3	`mergeDim`
Heterogene roll-up Beziehungen	
MERGE DIMENSIONS c1.kunde AS d3, c2.kunde AS d4 INTO c5.kunde AS d5 (RELATE d3.svnr, d4.kunde WHERE d3.svnr=d4.kunde USING HIERARCHY OF d3)	$\Omega_{'12334010180' \mapsto 'FullCom'}(kunde)$

Tabelle 5.2: SQL-MDi-Anweisungen vs. DA/FA-Operatoren 2/2

5.5 Zusammenfassung

In diesem Kapitel wurde demonstriert, wie mit der multidimensionalen Abfragesprache SQL-MDi die im Abschnitt 3.3.3 diskutierten Heterogenitäten eliminiert werden können. Darüber hinaus wurden der Sinn und Zweck der SQL-MDi zugrunde liegenden Dimension-Algebra (DA) und Fact-Algebra (FA), deren Operatoren für die unterschiedlichen SQL-MDi-Anweisungen sowie der Aufbau einer Operator-Baumstruktur erläutert. Da nicht alle Operatoren voneinander unabhängig angewendet werden können, wurden für solche Operatoren Einfüge-Regeln definiert, welche eine vollständige Verarbeitung der Baumstruktur ermöglichen. Zusätzlich wurden die zwischen einzelnen Operatoren möglichen logischen Konflikte mit der Abbildung 5.3 und den zugehörigen Erläuterungen präsentiert.

SQL-MDi zeichnet sich dadurch aus, dass alle im Abschnitt 3.3.3 vorgestellten Heterogenitäten mit einer Abfrage eliminiert und ein globaler, instanziierter Würfel erzeugt werden kann. Diese Mächtigkeit erreicht SQL-MDi durch die zugrunde liegende DA/FA, deren Operatoren für die Auflösung der Heterogenitäten auf Schema- und Instanzebene konzipiert wurden. Nach bestem Wissen des Autors ist SQL-MDi mit der DA/FA die erste Abfragesprache, welche die Integration von DWS im Rahmen eines FDWS derart umfassend unterstützt.

Teil III

Parser-Implementierung

Kapitel 6

Grundlagen des Übersetzerbaus

Das „Parsing“ (= Syntaxanalyse) ist eine Teilaufgabe bei der Entwicklung eines Übersetzers für Programmiersprachen. Ein Parser stellt fest, ob ein Input (z.B. Anweisungen in einer Programmiersprache) der Syntaxdefinition der Sprache entspricht [AU72]. In diesem Kapitel wird die für die Entwicklung der Parser-Komponente relevante Theorie behandelt, insbesondere die Grundlagen über formale Sprachen und Automaten (vgl. Abschnitt 6.1), welche für die Spezifikation der SQL-MDi-Syntax erforderlich sind, sowie die für die Parser-Entwicklung relevanten Übersetzungsphasen „lexikalische Analyse“ (vgl. Abschnitt 6.2.2), „Syntaxanalyse“ (vgl. Abschnitt 6.2.3) und „Semantikverarbeitung“ (vgl. Abschnitt 6.2.4).

6.1 Formale Sprachen und Automaten

Die Theorie der formalen Sprachen befasst sich mit der Analyse von Zeichenketten. Zeichenketten, welche die gleichen Struktureigenschaften aufweisen, werden zu Sprachen zusammengefasst, welche wiederum Sprachklassen zugeordnet werden können. Die Definition einer Sprache erfolgt mit Grammatiken oder Automaten, wodurch Sprachen, Grammatiken und Automaten eng miteinander in Beziehung stehen. [Rec06] Dieser Abschnitt dient der Erläuterung der so genannten regulären und kontextfreien Grammatiken, welche für die Beschreibung der Syntax von Programmiersprachen meist verwendet werden [HMU02]. Zuvor ist es jedoch notwendig einige Grundbegriffe zu klären.

6.1.1 Grundbegriffe

Im Folgenden werden die Begriffe „Zeichenketten und Sprachen“, „Grammatiken“, „EBNF“ und „Chomsky Hierarchie“ in Anlehnung an [Rec06] erläutert.

6.1.1.1 Zeichenketten und Sprachen

Das grundlegende Element ist das Alphabet V, welches eine endliche Menge von Zeichen umfasst, z.B. $V = \{a, b, c\}$. Zeichenketten werden durch das Hintereinanderschreiben von Zeichen gebildet, z.B. aa, ac, bac. Eine leere Zeichenkette wird durch ϵ dargestellt. Die Menge V^+ über dem Alphabet V ist die Menge aller nicht-leeren Ketten, welche sich aus Elementen von V bilden lassen; die Menge V^* ist die Menge aller Zeichenketten inklusive ϵ. Eine formale Sprache L über dem Alphabet V ist eine Teilmenge von $V^* : L \subseteq V^*$, deren Elemente als Sätze bezeichnet werden.

6.1.1.2 Grammatiken

Grammatiken bestehen aus einer Menge von (Produktions)Regeln, welche zwei Arten von Symbolen enthalten: Terminalsymbole (kommen in den Zeichenketten selbst vor) und Nonterminalsymbole (dienen der Strukturbeschreibung).

Definition 6.1. Eine Grammatik G ist ein Quadrupel der Form (V_N, V_T, P, S) mit

- V_N = Menge der Nonterminalsymbole,
- V_T = Menge der Terminalsymbole,
- P = Menge der Produktionsregeln $\alpha \rightarrow \beta$, wobei $\alpha \in V^* V_N V^*$ und $\beta \in V^*$
- S = Startsymbol aus V_N,
- $V_N \cap V_T = \oslash$, $V_N \cup V_T = V$.

Beispiel 6.1: Bei folgender Grammatik handelt es sich um eine rekursive Grammatik (aufgrund der Produktionsregel $B \rightarrow b|cB$), wodurch unendlich viele Zeichenketten beschrieben werden können, z.B. ab, acb, accb, acccb, etc.

$G = (\{S, A, B\}, \{a, b, c\}, \{S \rightarrow A, A \rightarrow aB, B \rightarrow b|cB\}, S)$.

6.1.1.3 Erweiterter Backus-Naur-Formalismus (EBNF)

Zur Beschreibung der Syntax von Programmiersprachen wird häufig der „Erweiterte Backus-Naur-Formalismus“ (EBNF) [Wir85] verwendet, welcher folgendermaßen charakterisiert ist:

- Terminalsymbole, die sich selbst bedeuten (Literale) in Anführungszeichen,
- Nonterminalsymbole: Wörter mit Bedeutung,
- Trennung von Regelseiten mit '=',
- Trennung von Alternativen mit '|',

- Regelende mit '.',
- Optionssymbol [A] bedeutet ϵ|A,
- Wiederholungssymbol {A} bedeutet ϵ|A|AA|AAA|AAAA|...,
- runde Klammern dienen der Zusammenfassung von Ausdrücken.

Beispiel 6.2: Die Grammatik des Beispiels 6.1 lässt sich mit dem EBNF folgendermaßen darstellen:

S = A.
A = "a"B.
B = "b"|"c"B.

6.1.1.4 Chomsky-Hierarchie

Die unterschiedlichen Typen von Grammatiken können mit Hilfe der Chomsky-Hierarchie [AO83] kategorisiert gegliedert werden.

Definition 6.2. Eine Grammatik $G = (V_N, V_T, P, S)$ gemäß Definition 6.1 heißt

- vom Typ 0 oder unbeschränkt, wenn alle Regeln gemäß der Definition 6.1 aufgebaut sind;
- vom Typ 1 oder kontextsensitiv, wenn alle Regeln die Form $\alpha A\beta \rightarrow \alpha\gamma\beta$ mit $\gamma \neq \epsilon$ aufweisen;
  $S \rightarrow \epsilon$ ist erlaubt, S darf jedoch auf keiner rechten Regelseite stehen;
- vom Typ 2 oder kontextfrei, wenn alle Regeln die Form $A \rightarrow \gamma$ haben;
- vom Typ 3 oder regulär/rechtslinear, wenn alle Regeln von einer der folgenden Formen sind: $A \rightarrow aB$, $A \rightarrow B$, $A \rightarrow a$, $A \rightarrow \epsilon$.

wobei $A, B \in V_N$, $a \in V_T$ und $\alpha, \beta, \gamma \in (V_N \cup V_T)^*$

Da die Informatik für die Definition der Syntax von Programmiersprachen oder von anderen strukturierten Sprachen, wie SQL und XML, vor allem reguläre und kontextfreie Sprachen bzw. Grammatiken (Typ 3 und Typ 2 der Chomsky-Hierarchie) verwendet, werden diese im Folgenden detaillierter behandelt.

6.1.2 Reguläre Sprachen und ihre Beschreibungsformen

Reguläre Sprachen werden beim Übersetzerbau vor allem für die Spezifikation der Komponente zur „lexikalischen Analyse“ (vgl. Abschnitt 6.2.1) verwendet, wo Bezeichner, Schlüsselwörter oder verschiedene Arten von Zahlen definiert werden [HMU02]. Im Folgenden werden drei Ansätze zur Beschreibung regulärer Sprachen vorgestellt.

6.1.2.1 Reguläre Grammatiken

Wie aus der Definition 6.2 ersichtlich, können reguläre (rechtslineare) Grammatiken in den rechten Regelseiten höchstens ein Nonterminalsymbol, und zwar am Ende, enthalten.

Beispiel 6.3: Die folgende Grammatik definiert eine reguläre Sprache für Bezeichner, wobei ein solcher aus einer beliebigen Anzahl von Buchstaben b und Ziffern z bestehen kann, jedoch mit einem Buchstaben beginnen muss:

$G = (\{S, R\}, \{b, z\}, \{S \rightarrow bR, R \rightarrow bR|zR|\epsilon\}, S)$.

6.1.2.2 Reguläre Ausdrücke

Reguläre Ausdrücke stellen ein deklaratives Verfahren zur Beschreibung regulärer Sprachen dar und sind in der Informatik sehr gebräuchlich, z.B. für Befehle im UNIX-Betriebssystem [HMU02]. Ein regulärer Ausdruck *RegExpr* über V_T kann mit der EBNF-Syntax folgendermaßen beschrieben werden: [Rec06]

RegExpr = RegTerm {"+" RegTerm}|∅.
RegTerm = RegFact {RegFact}"ϵ".
RegFact = a["*"]|"(" RegExpr ")"["*"].

wobei ∅, $\{\epsilon\}$, und {a}, für a $\in V_T$, reguläre Mengen über V_T sind.

Beispiel 6.4: Der folgende reguläre Ausdruck beschreibt die reguläre Sprache der Bezeichner der Grammatik des Beispiels 6.3:

Bezeichner = b(b + z)*

6.1.2.3 Endliche Automaten

Ein endlicher Automat besitzt eine Menge von Zuständen, wobei der Übergang von einem Zustand in den nächsten (= Steuerung) von externen Inputs (Zeichen einer Zeichenkette) bestimmt wird. Eine wichtige Unterscheidung zwischen endlichen Automaten ist, ob die Steuerung deterministisch ist, d.h. dass der Automat zu einem Zeitpunkt nur einen Folgezustand haben kann, oder nichtdeterministisch ist, d.h. dass mehrere Folgezustände möglich sind. Man spricht in diesem Zusammenhang von deterministischen (DEA) bzw. nichtdeterministischen, endlichen Automaten (NEA).

Definition 6.3. Ein DEA ist ein Quintupel (Z, V_T, δ, z_1, F) mit: [HMU02]

- einer endlichen Menge von Zuständen Z,
- einer endlichen Menge von Eingabezeichen V_T,
- einer Übergangsfunktion $\delta(z, a)$ mit $z \in Z$ und $a \in V_T$, welche ein Eingabezeichen und den aktuellen Zustand als Parameter besitzt und einen neuen Zustand liefert,
- einem Startzustand $z_1 \in Z$,
- einer Menge von Endzuständen $F \subseteq Z$.

Eine gelesene Zeichenkette gilt als durch einen DEA akzeptiert, wenn sich dieser in einem Endzustand befindet.

Zusätzlich kann für die Darstellung der Zustandsübergänge eine „Konfiguration“ angegeben werden. Eine Konfiguration (z, α) besteht aus dem aktuellen Zustand z und dem noch ungelesenen Teil der Eingabezeichen α, wobei das äußerst links stehende Zeichen von α als nächstes gelesen wird. [Rec06]

Beispiel 6.5: Deterministischer, endlicher Automat für die reguläre Sprache der Bezeichner des Beispiels 6.3:

$DEA = (\{z_1, z_2\}, \{b, z\}, \delta, z_1, \{z_2\})$
mit $\delta(z_1, b) = z_2$, $\delta(z_2, b) = z_2$, $\delta(z_2, z) = z_2$

Die Erkennung der Zeichenkette bbzzb führt zu folgenden Zustandsübergängen (Konfigurationen):
(z_1,bbzzb) $\mapsto$ (z_2,bzzb) $\mapsto$ (z_2,zzb) $\mapsto$ (z_2,zb) $\mapsto$ (z_2,b) $\mapsto$ (z_2,ϵ)

Dieser DEA lässt sich auch durch folgenden Zustandsgraphen veranschaulichen (vgl. Abbildung 6.1):

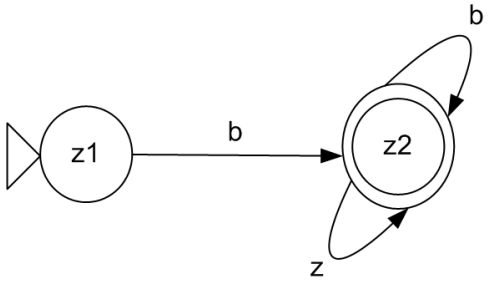


Abbildung 6.1: Zustandsgraph eines deterministischen, endlichen Automaten (vgl. [HMU02])

Die Bestandteile eines NEA gleichen denen eines DEA mit der Ausnahme, dass die Übergangsfunktion statt einem Zustand eine Menge von Zuständen liefern kann. Die Fähigkeit des NEA über mehrere Folgezustände zu verfügen, kann man sich als Parallelverfolgung der verschiedenen möglichen Wege vorstellen. DEA und NEA sind jedoch äquivalent hinsichtlich ihrer Eigenschaft reguläre Sprachen zu akzeptieren. [HMU02]

6.1.3 Kontextfreie Sprachen und ihre Beschreibungsformen

Kontextfreie Sprachen werden beim Übersetzerbau für die Spezifikation der Syntaxanalyse (vgl. Abschnitt 6.2.1) eingesetzt. Dank ihnen wurde die ehemals zeitaufwändige Tätigkeit der Parser-Implementierung zur Routine [HMU02]. Im Folgenden werden zwei Ansätze zur Beschreibung kontextfreier Sprachen vorgestellt.

6.1.3.1 Kontextfreie Grammatiken

Kontextfreie Grammatiken basieren auf zentralrekursiven Regeln der Form $A \rightarrow \alpha A \beta$ (direkt) oder $A \rightarrow \alpha B, B \rightarrow \gamma A \delta$ (indirekt) mit $A, B \in V_N$ und $\alpha, \beta, \gamma, \delta \in (V_T \cup V_N)$ [Rec06].

Die Regeln einer kontextfreien Grammatik können auf zwei Arten angewendet werden, um zu bestimmen, ob eine bestimmte Zeichenkette ein gültiger Satz einer Sprache ist. Bei der „rekursiven Inferenz“ wird bottom-up vorgegangen, d.h. als korrekt identifizierte, gelesene Zeichen (= Terminalsymbole) werden rekursiv zu immer gröberen Nonterminalsymbolen zusammengefasst, bis das Startsymbol erreicht wird.

Bei der zweiten Verwendungsart, der „Ableitung“, wird ein top-down Ansatz verfolgt, indem, ausgehend vom Startsymbol, Nonterminalsymbole durch die rechten Seiten der entsprechenden Regeln ersetzt werden bis der entstehende Satz der Zeichenkette entspricht. Bezüglich der Art der Ableitung können linksseitige und rechtsseitige Ableitungen unterschieden werden. Bei der linksseitigen Ableitung wird in jedem Schritt das äußerst linke Nonterminalsymbol durch das (die) Terminalsymbol(e) der entsprechenden Regel ersetzt, wogegen bei der rechtsseitigen Ableitung in jedem Schritt das äußerst rechte Nonterminalsymbol ersetzt wird. [HMU02]

Beispiel 6.6: Die folgende Grammatik definiert eine kontextfreie Sprache über $V = \{a, b\}$, in deren Sätzen die Anzahl der a doppelt so groß wie die Anzahl der b ist und die a vor den b stehen müssen (z.B. aab, aaaabb, aaaaaabbb, etc.):

$G = (\{S\}, \{a, b\}, \{S \rightarrow \epsilon, S \rightarrow aaSb\}, S)$.

Weitere Informationen zu kontextfreien Sprachen bzw. Grammatiken werden im Abschnitt 6.2.3 im Rahmen der Behandlung des Themas „Syntaxanalyse“ präsentiert.

6.1.3.2 Kellerautomaten

Für die Erkennung von kontextfreien Sprachen reichen endliche Automaten nicht aus, da zusätzlich ein Zwischenspeicher für Zeichen (Keller bzw. Stack) erforderlich ist. Kellerautomaten entsprechen endlichen Automaten, erweitern jedoch diese um einen solchen Zwischenspeicher.

Definition 6.4. Ein Kellerautomat ist ein Septupel $(Z, V_T, V, \delta, z_1, S, F)$ und unterscheidet sich gegenüber einem endlichen Automaten (vgl. Definition 6.3) folgendermaßen: [HMU02]

- Er umfasst zusätzlich eine endliche Menge von Kellerzeichen V.
- Die Übergangsfunktion $\delta(z, a, k)$, liefert aufgrund des aktuellen Zustands z, des nächsten Eingabezeichens a und des nächsten Kellerzeichens k einen neuen Zustand (bei endlichen Automaten erfolgt der Zustandsübergang nur aufgrund des aktuellen Zustands und des nächsten Eingabezeichens). Die Ausgabe von δ ist eine Menge von Paaren (x, γ), wobei x den neuen Zustand und γ für die Menge der Kellerzeichen steht, welche k auf dem oberen Ende des Kellers ersetzt. Das heißt, wenn z.B. $\gamma = \epsilon$, wird k entfernt („pop“); wenn $\gamma = k$, bleibt der Keller unverändert, und wenn $\gamma = \text{lm}$, dann wird k durch m ersetzt und l oben auf den Keller gelegt („push“).
- Er umfasst zusätzlich ein Kelleranfangszeichen $S \in V$.

Analog zu endlichen Automaten, kann auch für Kellerautomaten eine Konfiguration zur Darstellung von Zustandsübergängen definiert werden. Eine Konfiguration $(z, k.\alpha)$ beschreibt den aktuellen Zustand z, den Kellerinhalt k sowie den noch ungelesenen Teil der Eingabezeichen α. Außerdem können bei Kellerautomaten, wie bei endlichen Automaten, deterministische und nichtdeterministische unterschieden werden. Ein Kellerautomat ist deterministisch, wenn folgende zwei Bedingungen gelten: [Rec06]

1. Für ein beliebiges $z \in Z, k \in V$ und $a \in V_T$ existieren $\delta(z, \epsilon, k)$ oder $\delta(z, a, k)$, aber nicht beide.
2. Für jedes $z \in Z, k \in V$ und $b \in V_T \cup \{\epsilon\}$ enthält $\delta(z, b, k)$ höchstens einen Folgezustand.

Im Unterschied zu endlichen Automaten, kann ein Kellerautomat eine vollständig gelesene Zeichenkette auf zwei Arten akzeptieren: wenn sich der Kellerautomat in einem Endzustand befindet oder wenn der Keller leer ist.

Beispiel 6.7: Kellerautomat für die kontextfreie Grammatik des Beispiels 6.6:

$KA = (\{z_1, z_2, z_3\}, \{a, b\}, \{a, b, Z_0\}, \delta, z_1, Z_0, \{z_3\})$, wobei für δ folgende Regeln gelten:

1. $\delta(z_1, \text{a}, Z_0) = \{(z_2, Z_0)\}$. Befindet sich der Kellerautomat im Startzustand z_1 und wird ein a gelesen, folgt ein Übergang in den Zustand z_2 ohne den Keller zu verändern.
2. $\delta(z_2, \text{a}, Z_0) = \{(z_1, \text{a})\}$. Befindet sich der Kellerautomat im Zustand z_2 und wird ein a gelesen, folgt ein Übergang zurück in den Zustand z_1 sowie die Ablage von a oben auf dem Keller, wobei Z_0 darunter bleibt.
3. $\delta(z_1, \text{a}, \text{a}) = \{(z_2, \text{a})\}$. Befindet sich der Kellerautomat im Zustand z_1, wird ein a gelesen und befindet sich ein a oben auf dem Keller, folgt ein Übergang in den Zustand z_2 ohne den Keller zu verändern.
4. $\delta(z_2, \text{a}, \text{a}) = \{(z_1, \text{aa})\}$. Befindet sich der Kellerautomat im Zustand z_2, wird ein a gelesen und befindet sich ein a oben auf dem Keller, folgt ein Übergang zurück in den Zustand z_1 sowie die Ablage eines weiteren a oben auf dem Keller, wobei die anderen Kellerzeichen darunter bleiben.

5. $\delta(z_1, \text{b}, \text{a}) = \{(z_3, \epsilon)\}$. Befindet sich der Kellerautomat im Zustand z_1, wird ein b gelesen und befindet sich ein a oben auf dem Keller, folgt ein Übergang in den Zustand z_3 sowie die Entfernung des obersten a vom Keller.
6. $\delta(z_3, \text{b}, \text{a}) = \{(z_3, \epsilon)\}$. Befindet sich der Kellerautomat im Zustand z_3, wird ein b gelesen und befindet sich ein a oben auf dem Keller, erfolgt kein Zustandsübergang sondern nur die Entfernung des obersten a vom Keller.
7. $\delta(z_3, \epsilon, Z_0) = \{(z_3, \epsilon)\}$. Befindet sich der Kellerautomat im Zustand z_3, wurde die Zeichenkette vollständig gelesen (ϵ) und ist das Kelleranfangszeichen Z_0 das einzige (oberste) Kellerzeichen, so wird dieses entfernt und die Zeichenkette gilt aufgrund des leeren Kellers als akzeptiert.

Die Regeln 1 - 4 sorgen dafür, dass eine beliebige, gerade Anzahl von a vor einer Anzahl von b in der Zeichenkette vorkommt. Wird im Zustand z_1 ein b gelesen folgt ein Übergang zum Zustand z_3, wodurch sichergestellt wird, dass jetzt keine a mehr folgen dürfen, sondern nur mehr b (Regeln 5 - 6). Eine Zeichenkette wird mit der Regel 7 akzeptiert, wenn das Ende der Zeichenkette ϵ erreicht wird und Z_0 das einzige (oberste) Kellerzeichen ist, welches entfernt wird.

Die Erkennung der Zeichenkette aaaabb führt zu folgenden Zustandsübergängen (Konfigurationen):

$(z_1, Z_0 \text{ . aaaabb}) \mapsto (z_2, Z_0 \text{ . aaabb}) \mapsto (z_1, aZ_0 \text{ . aabb}) \mapsto (z_2, aZ_0 \text{ . abb}) \mapsto (z_1, aaZ_0 \text{ . bb}) \mapsto$
$(z_3, aZ_0 \text{ . b}) \mapsto (z_3, Z_0 \text{ . } \epsilon) \mapsto (z_1, \epsilon \text{ . } \epsilon)$.

Nachdem nun die für den Übersetzerbau relevante Theorie der formalen Sprachen und Automaten behandelt wurde, folgt im nächsten Abschnitt die Erläuterung der für diese Diplomarbeit relevanten Teilbereiche des Übersetzerbaus.

6.2 Übersetzerbau

Ein Übersetzer ist „ein Programm, das ein in einer bestimmten Sprache, der Quellsprache, geschriebenes Programm liest und in ein äquivalentes Programm einer anderen Sprache, der Zielsprache, übersetzt“ [ASU88]. Es gibt eine Vielfalt von Übersetzern für unterschiedlichste Quell- und Zielsprachen. Quellsprachen können traditionelle Programmiersprachen sein, wie Fortran oder C, sowie Spezialsprachen, die für spezifische Domänen verwendet werden; Zielsprache kann eine andere Programmiersprache oder eine Maschinensprache eines bestimmten Rechners sein [ASU88]. Das Übersetzungsproblem lässt sich auch allgemeiner charakterisieren: die Analyse und Transformation „eines Stroms von Informationen, der eine komplexe Struktur hat und zwischen dessen Elementen komplexe Beziehungen bestehen“ [Kas99]. Aufgrund dieser Definition lassen sich somit auch beliebige Zeichenketten, deren Struktur mit einer Grammatik beschrieben werden kann, in eine bestimmte Zielsprache transformieren.

Das Übersetzungsproblem lässt sich in eine Menge von Übersetzungsphasen gliedern, welche im Folgenden vorgestellt werden.

6.2.1 Übersetzungsphasen

Abbildung 6.2 zeigt die verschiedenen Übersetzungsphasen, welche einem Analyseteil (bestimmt durch die Quellsprache) oder einem Syntheseteil (bestimmt durch die Zielsprache) zugeordnet werden können:

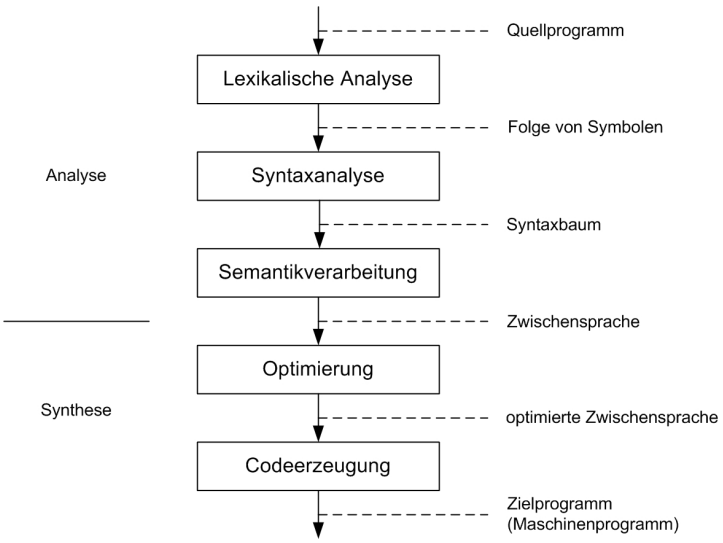


Abbildung 6.2: Übersetzungsphasen (vgl. [Mös06a])

Im Zuge der lexikalischen Analyse wird das Quellprogramm in eine Folge von Symbolen zerlegt. Die Syntaxanalyse prüft die syntaktische Korrektheit des Quellprogramms, indem die vom lexikalischen Analysator gelieferten Symbolfolgen in syntaktische Einheiten (gemäß einer definierten Grammatik) zerlegt werden. In der Semantikverarbeitung werden semantische Regeln (Kontextbedingungen) geprüft und Datenstrukturen bzw. eine Zwischensprache (vgl. Abschnitt 6.2.4.3) aufgebaut. In der Optimierungsphase wird versucht die Laufzeit- und Speicherplatzeffizienz des Zwischencodes zu verbessern. Durch die Codeerzeugung erfolgt letztlich die Transformation in die Zielsprache; handelt es sich dabei um eine Maschinensprache, müssen deren Hardwarespezifika (z.B. Speicheradressierung) berücksichtigt werden. [Mös06a]

Für die im Zuge der Diplomarbeit entwickelte Parser-Komponente sind Kenntnisse über die Phasen „lexikalische Analyse“, „Syntaxanalyse“ und „Semantikverarbeitung“ erforderlich, weshalb diese im Anschluss detaillierter behandelt werden. Die vom entwickelten Parser generierte Operator-Baumstruktur kann gemäß Abbildung 6.2 als Zwischensprache betrachtet werden. Diese wird von einer Prozessor-Komponente in die Zielsprache, in SQL-MDi-Anweisungen für die Komponentensysteme, transformiert.

6.2.2 Lexikalische Analyse

Die Hauptaufgabe der Komponente, welche die lexikalische Analyse realisiert (= Scanner), ist das Einlesen der Eingabezeichen, aus denen eine Folge von Symbolen erzeugt wird. Dabei wird für jedes Symbol ein numerischer Symbolcode, ein Symbolwert und die Symbolposition (für Fehlermeldungen) ermittelt [Mös06a]. Diese Symbolfolgen werden an den Parser zur Syntaxprüfung übergeben. Darüber hinaus ist das Überlesen von bedeutungslosen Zeichen (z.B. Leerzeichen, Zeilenumbrüche oder Tabulatoren) und von Kommentaren eine wichtige Aufgabe des Scanners um die Implementierung der Syntaxanalyse zu vereinfachen. Dies ist auch der Hauptgrund für die Trennung von Scanner und Parser [App98].

6.2.2.1 Spezifikation von Symbolen

Die Spezifikation von Symbolen, welche z.B. Namen, Zahlen oder Schlüsselwörter sein können, erfolgt unter Nutzung der Tatsache, dass sie mit einer regulären Sprache beschrieben werden können. Wie im Abschnitt 6.1.2 erläutert, können reguläre Sprachen durch reguläre Grammatiken, reguläre Ausdrücke oder endliche Automaten definiert werden. Meist werden bei der Erstellung einer Grammatik für eine Programmiersprache reguläre Ausdrücke für die Symbol-Spezifikationen verwendet. [Kas99]

Beispiel 6.8: Spezifikation des Symbols „Bezeichner“ (vgl. Beispiele 6.3 - 6.5):

Buchstabe = a|..|z|A|..|Z
Ziffer = 0|1|2|3|4|5|6|7|8|9|

Bezeichner = Buchstabe (Buchstabe | Ziffer)*

6.2.2.2 Erkennung von Symbolen

Die Erkennung von Symbolen basiert auf den im Abschnitt 6.1.2.3 erläuterten endlichen Automaten. Die Erstellung endlicher Automaten für die Erkennung von Symbolen kann als (gedanklicher) Zwischenschritt bei der Implementierung eines Scanners betrachtet werden.

Beispiel 6.9: Listing 6.1 stellt einen Ausschnitt aus einer Java-Implementierung eines Scanners für die Erkennung von Bezeichnern, gemäß Beispiel 6.8, dar.

```
1  Token nextToken(){
2    Token sym = new Token(); /* Symbol (Bezeichner) */
3    int state = 1;           /* Startzustand z1 */
4    char c = '';
5      while(true){
6        switch(state){
7          case 1: c = nextChar();  /* 1. Zeichen muss Buchstabe sein */
8                  if(isLetter(c)){
9                    state=2;       /* Zustandsübergang z2 */
10                   sym.add(c);    /* Zeichen zu Token hinzufügen */
11                 }else{
12                   state=fail();
13                 }
14                 break;
15         case 2: c= nextChar();     /* Buchstabe oder Ziffer */
16                 if(isLetter(c)){
17                   state=2;
18                   sym.add(c);
19                 }else if(isDigit(c)){
20                   state=2;
21                   sym.add(c);
22                 }else if(isEOL(c)){ /* Ende der Zeichenkette */
23                   return sym;       /* Rückgabe des Symbols */
24                 }else{
25                   state=fail();
26                 }
27                 break;
28         default: return sym=null;   /* bei Fehler */
29       }
30   }
31 }
```

Listing 6.1: Scanner-Implementierung

Die Methode `nextToken()` demonstriert, wie Symbole (`Token sym`) aus einzelnen Zeichen konstruiert werden können. Der endliche Automat aus Abbildung 6.1 wird mit Hilfe einer Endlosschleife und einer `switch` Anweisung, in welcher die Zustandsprüfungen und Zustandsübergänge erfolgen, realisiert, d.h. die Anweisung `case 1:` repräsentiert den Zustand z1, die Anweisung `case 2:` den Zustand z2. Wird ein Zeichen korrekt erkannt (z.B. ein Buchstabe im Zustand z1), wird dieses mit der Anweisung `sym.add(c)` zum Symbol hinzugefügt und ggf. ein Zustandsübergang durchgeführt (`state=2`). Wird ein in einem Zustand nicht gültiges Zeichen gelesen (z.B. eine Ziffer im Zustand z1) wird mit der Methode `state=fail()` die Fehlerbehandlung angestoßen. Wird ein EOL (end of line) Symbol im Zustand z2 erkannt, wird das erkannte Symbol zurückgegeben.

6.2.3 Syntaxanalyse

Jede Programmiersprache oder sonstige domänenspezifische Sprache umfasst Regeln, welche beschreiben, wie die syntaktische Struktur wohlgeformter Programme bzw. von Sätzen aussehen muss. Diese Regeln lassen sich mit den im Abschnitt 6.1.3.1 behandelten kontextfreien Grammatiken beschreiben. Die Hauptaufgabe der Syntaxanalyse bzw. der Parser-Komponente eines Übersetzers ist die Prüfung der syntaktischen Korrektheit des Quellprogramms bzw. der Quell-Zeichenkette. Dabei prüft der Parser, ob die vom Scanner gelieferten Symbolfolgen mit

der Grammatik der Quellsprache erzeugt werden können und generiert im gegensätzlichen Fall eine aussagekräftige Fehlermeldung. Die für diesen Zweck vorwiegend eingesetzten Methoden werden als „Top-down-Analyse“ und „Bottom-up-Analyse“ bezeichnet. [ASU88]

6.2.3.1 Top-down-Analyse

Bei der Top-down-Analyse wird ein Syntaxbaum von oben nach unten aufgebaut. Dabei werden, ausgehend vom Startsymbol der Grammatik, Nonterminalsymbole durch die rechte Seite ihrer Regeln ersetzt bis der entstehende Satz der Input-Zeichenkette entspricht. Das Generieren des Syntaxbaums erfolgt dabei durch links-Ableitungen, wobei das am weitesten links stehende Nonterminalsymbol ersetzt wird. Tritt der Fall auf, dass zwischen mehreren Alternativen gewählt werden muss, kann durch Vorgriff um k Zeichen eine Entscheidung getroffen werden. Die von der Top-down-Analyse verarbeitbare Klasse von Grammatiken wird als LL(k) bezeichnet. [ASU88]

Definition 6.5. „Eine Grammatik wird LL(k) genannt, wenn man bei der absteigenden Syntaxanalyse (Top-down-Analyse) von links nach rechts in jeder Situation, in der man zwischen Alternativen wählen muss, durch den Vorgriff um k Zeichen entscheiden kann, welche die richtige ist.“ [Rec06]

Die in der Praxis wichtigste Ausprägung dieser Grammatik ist LL(1), also mit einem Vorgriffssymbol (lookahead symbol) [ASU88]. Beispiel 6.10 zeigt, wie der Syntaxbaum bei der Top-down-Analyse für die Grammatik aus Beispiel 6.6 aufgebaut wird.

Beispiel 6.10: Der Syntaxbaum für die Input-Zeichenkette aaaabb wird in den in Abbildung 6.3 gezeigten Schritten aufgebaut, wobei das Vorgriffssymbol sowie das nächste zu erkennende Nonterminalsymbol durch einen Kreis gekennzeichnet sind.

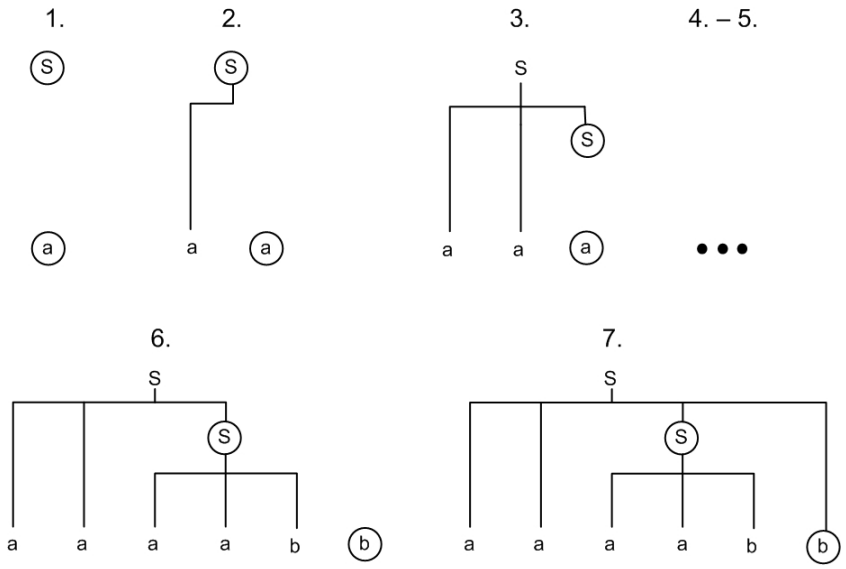


Abbildung 6.3: Aufbau eines Syntaxbaums bei der Top-down-Analyse

Listing 6.2 zeigt eine Java-Implementierung der Grammatik des Beispiels 6.6 (wie sie der Übersetzer-Generator Coco/R [LMW] erzeugt, vgl. Abschnitt 6.2.5), welche den Syntaxbaum entsprechend der Abbildung 6.3 aufbaut.

```
1  void parse() {
2    if (la == ''a'') {
3      Get();            /* la = nächstes Zeichen */
4      Expect(''a'');    /* if(la!=aktuelles Zeichen) Error() */
5      parse();
6      Expect(''b'');
7    }else if (la.kind == EOL) {  /* Ende der Zeichenkette */
8      Get();
9    }else Error();      /* Fehlermeldung */
10 }
```

Listing 6.2: Top-down Parser-Implementierung

Die Variable `la` stellt das lookahead symbol dar. Hat `la` den Wert a, wird `la` das nächste Input-Zeichen mit der Methode `Get()` zugewiesen. Mit der Methode `Expect(x)` wird das nächste erwartete Terminalsymbol erkannt, indem `la` mit dem entsprechenden Parameter (z.B. a) verglichen wird. Treten andere Zeichen außer a oder b auf wird mit der Methode `Error()` eine Fehlermeldung ausgegeben. Die Rekursion endet, wenn ein `EOL` (end of line) Symbol oder ein Fehler auftritt.

6.2.3.2 Bottom-up-Analyse

Bei der Bottom-up-Analyse wird ein Syntaxbaum von unten nach oben aufgebaut. Die Grundidee dieses Analyseverfahrens besteht darin, dass eine Zeichenkette auf das Startsymbol der Grammatik „reduziert“ wird. Das heißt, bei jedem Reduktionsschritt wird eine Folge von Symbolen, welche mit der rechten Seite einer Regel übereinstimmt, durch das Nonterminalsymbol der linken Regelseite ersetzt. Dies wird solange wiederholt, bis die gesamte Zeichenkette zum Syntaxbaum mit dem Startsymbol an der Wurzel reduziert wurde. Tritt der Fall auf, dass zwischen mehreren Alternativen gewählt werden muss, kann durch Vorgriff um k Zeichen eine Entscheidung getroffen werden. Die von der Bottom-up-Analyse verarbeitbare Klasse von Grammatiken wird als LR(k) bezeichnet. [ASU88]

Definition 6.6. „Eine Gramatik wird LR(k) genannt, wenn bei der aufsteigenden Syntaxanalyse (Bottom-up-Analyse) von links nach rechts in jeder Situation durch Betrachtung des gesamten bisher gelesenen Teils der Zeichenkette oder der durch Reduktion aus ihr entstandenen Satzform und von k Vorgriffszeichen eindeutig festgestellt werden kann, ob der gelesene Teil eine vollständige Phrase (rechte Seite einer Regel) enthält, und wenn ja, wie lang sie ist und zu welchem Nonterminalsymbol sie reduziert werden muss.“ [Rec06]

Analog zu LL(k)-Grammatiken, reicht in der Praxis meist ein Vorgriffssymbol (lookahead symbol) aus (= LR(1)-Grammatik). Beispiel 6.11 zeigt, wie ein Syntaxbaum bei der Bottom-up-Anaylse für die Grammatik des Beispiels 6.6 generiert wird.

Beispiel 6.11: Der Syntaxbaum für die Input-Zeichenkette aaaabb wird in den in Abbildung 6.4 gezeigten Schritten aufgebaut.

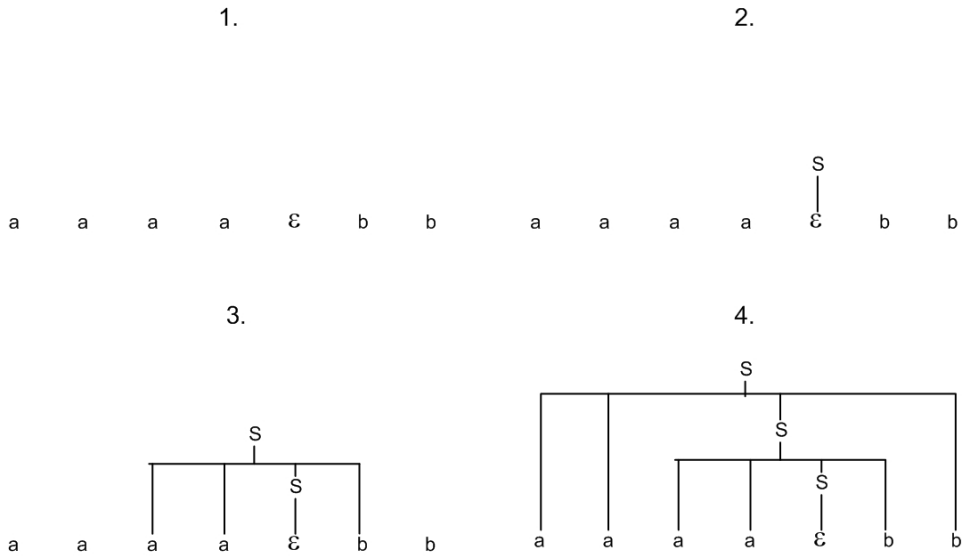


Abbildung 6.4: Aufbau eines Syntaxbaums bei der Bottom-up-Analyse

Auf die Darstellung einer Implementierung der Bottom-up-Analyse wird hier verzichtet. Es sei jedoch

angemerkt, dass die Bottom-up-Analyse der Arbeitsweise eines Kellerautomaten (vgl. Abschnitt 6.1.3.2) ähnelt. Die konkrete Implementierung erfolgt oft tabellengesteuert, d.h. der Kellerautomat wird als Tabelle realisiert, die für jeden Zustand und jedes Eingabesymbol die auszuführende Aktion angibt. [Mös06a] [DP82]

6.2.3.3 Fehlerbehandlung

Im Zuge der Fehlerbehandlung sollen Syntaxfehler an den Benutzer gemeldet werden. Es ist jedoch nicht wünschenswert, dass der Parser einen Fehler meldet und anschließend die Arbeit beendet. Dies hätte zur Folge, dass der Benutzer gezwungen ist, jeden Syntaxfehler einzeln zu beheben und die Analyse neu zu starten. Ziel der Fehlerbehandlung ist daher die Anzeige von Syntaxfehlern und die Fortsetzung der Analyse durch den Parser. Die in den Abschnitten 6.2.3.1 und 6.2.3.2 betrachteten Analyseverfahren, Top-down und Bottom-up, können Fehler bereits an der Stelle ihres Auftretens erkennen. [Kas99] Wie im Listing 6.2 ersichtlich, wird beim Auftreten eines ungültigen Zeichens sofort eine Fehlermeldung erzeugt.

Da sich der Parser nach dem Auftreten eines Fehlers in einem ungültigen Zustand befindet, ist es erforderlich den internen Zustand des Parser zu verändern um die Analyse fortzusetzen. Gängige Verfahren zur Fehlerbehandlung sind: [Mös06a]

- **Panik-Modus:** Ab dem Melden des ersten Fehlers wird die Syntaxanalyse abgebrochen. Diese rudimentäre Fehlerbehandlung ist nicht wünschenswert.
- **Wiederaufsatz mit allgemeinen Fangsymbolen:** Im Fehlerfall wird die Symbolfolge solange überlesen, bis ein Symbol auftritt, welches an der Fehlerstelle erwartet wird oder das Nachfolger der gerade in Arbeit befindlichen Regel (Nonterminalsymbol) ist (das EOL Symbol ist immer ein Nachfolger).
- **Wiederaufsatz mit speziellen Fangsymbolen:** Im Gegensatz zur vorherigen Variante erfolgt die Synchronisation hier nur an bestimmten Stellen, an denen selten vorkommende Symbole erwartet werden (z.B. Schlüsselwörter).

6.2.4 Semantikverarbeitung

Die Semantikverarbeitung betrifft die Prüfung kontextabhängiger Eigenschaften von Elementen eines Programms sowie den Aufbau von Datenstrukturen bzw. die Transformation in eine Zwischensprache. Die Spezifikation der kontextabhängigen Eigenschaften erfolgt häufig mit Symboltabellen und attributierten Grammatiken. Zu den zu prüfenden kontextabhängigen Eigenschaften zählen Typprüfungen und Bezeichneridentifikation. [Kas99] [App98]

6.2.4.1 Symboltabellen

Symboltabellen werden unter anderem verwendet, um Bezeichner auf ihre Typen abzubilden. Das heißt, werden Typ-, Variablen- oder Methodendeklarationen verarbeitet, werden diese Bezeichner mit ihrem Kontext in eine Symboltabelle eingetragen; werden Verwendungen von Bezeichnern identifiziert, folgt ein „lookup“ in den Symboltabellen um den Kontext abzufragen. Die Implementierung der Symboltabelle erfolgt häufig als dynamische Datenstruktur (z.B. als Hash-Tabelle), in welcher jedes Symbol durch einen Objektknoten realisiert wird.

Beispiel 6.12: Listing 6.3 zeigt die Java-Implementierung einer Symboltabelle als Hash-Tabelle:

```
1  public class Symbol{
2    private String name;
3    private Symbol(String n){
4      name = n;
5    }
6    private static java.util.Dictionary dict =
7    new java.util.Hashtable();
8
9    public String toString() {
10     return name;
11   }
12
13   public static Symbol symbol(String n){
14     String u = n.intern();  /* eindeutiges String-Objekt */
15     Symbol s = (Symbol)dict.get(u);
16     if(s==null){
17       s = new Symbol(u);
18       dict.put(u,s);
19     }
20     return s;
21   }
22 }
```

Listing 6.3: Implementierung einer Symboltabelle [App98]

In der Methode `symbol(String n)` wird zunächst ein eindeutiges String-Objekt mit der String-Methode `intern()` generiert, für welches in der Hash-Tabelle das entsprechende Symbol abgefragt wird. Ist kein Symbol vorhanden, wird das aktuelle Symbol in die Hash-Tabelle eingefügt.

6.2.4.2 Attributierte Grammatiken

Attributierte Grammatiken erweitern kontextfreie Grammatiken (vgl. Abschnitt 6.1.3.1) um Attribute, Kontextbedingungen und semantische Aktionen um kontextabhängige Prüfungen durchführen zu können. Attribute werden verwendet, um den Datenaustausch zwischen Prozessen zu ermöglichen. Es werden zwei Arten von Attributen unterschieden: Eingangsattribute werden einem Nonterminalsymbol zu seiner Erkennung als Parameter mitgegeben und entsprechen Eingangsparametern; Ausgangsattribute entstehen bei der Erkennung eines Terminal- oder Nonterminalsymbols und entsprechen Ausgangsparametern. Semantische Aktionen dienen der Beschreibung von Verarbeitungsprozessen in den Produktionsregeln der Grammatik. [PP04]

Beispiel 6.13: Listing 6.4 zeigt eine attributierte Grammatik, welche das Verhältnis der in der Zeichenkette vorkommenden Anzahl der a zur Anzahl der b berechnet. Für die Darstellung der Attributierungen wird die Syntax der Grammatik-Spezifikation für den Übersetzer-Generator Coco/R [Mös06b] (vgl. Abschnitt 6.2.5) verwendet, da diese leicht verständliche Konstrukte für Attributierungen bietet.

```
1  Output                  (. int prop; .)
2   = Chars<out prop>      (. System.out.println
3                          (''a-to-b proportion: ''+prop); .).
4
5  Chars<out int prop>     (. int a_count=0; int b_count=0; .)
6  = (a                    (. a_count++; .)
7  |b                      (. b_count++; .)
8  )
9  {a                      (. a_count++; .)
10 |b                      (. b_count++; .)
11 }                       (. if(b_count!=0)prop = a_count/b_count; .)
12                         (. else prop = a_count; .).
```

Listing 6.4: Attributierte Grammatik mit Coco/R

Die Grammatik-Spezifikation für den Übersetzer-Generator Coco/R ähnelt der EBNF-Syntax (vgl. Abschnitt 6.1.1.3). Die Variablen zwischen < ... > stellen Attribute dar; in diesem Fall handelt es sich um Ausgangsattribute (Schlüsselwort `out`). Semantische Aktionen werden in `(. ... .)` definiert und an der Stelle innerhalb einer Produktionsregel platziert, wo die Aktion auch ausgeführt werden soll. In diesem Beispiel wurde nach jedem Auftreten eines Zeichens eine semantische Aktion platziert, welche einen zuvor deklarierten Zähler um 1 erhöht (`a_count` für a und `b_count` für b). Am Ende der Zeichenkette wird das Verhältnis `prop` berechnet und an die Regel `Output` zurück geliefert, die den Wert `prop` mit Hilfe einer entsprechenden semantischen Aktion ausgibt.

6.2.4.3 Zwischensprache

Die Erzeugung der Zwischensprache bildet den Abschluss der Analysephase und die Schnittstelle zur Synthesephase. Es geht darum, die Analyseergebnisse in eine geeignete Form zu transformieren um die folgenden Synthesephasen „Optimierung“ und „Codeerzeugung“ leichter bearbeiten zu können. Der im Zuge der Analyse entstehende Syntaxbaum enthält oft Konstrukte, welche nur für die Analyse, nicht aber für die Synthese benötigt werden. In der Zwischensprache werden daher überflüssige Konstrukte und Attribute entfernt um die Synthese zu erleichtern. [Kas99]

Darüber hinaus ist es oft sinnvoll, wenn man in der Zwischensprache Ausdrücke, welche mit unterschiedlichen Techniken auf der Zielmaschine implementiert werden, strukturell unterscheidet, d.h. spezielle Konstrukte für die entsprechenden Knoten des Syntaxbaums einführt. Entwickelt man die Zwischensprache unabhängig von einer bestimmten Quellsprache, kann der sie übersetzende Syntheseteil unverändert für Übersetzer verschiedener Quellsprachen verwendet werden. [Kas99]

6.2.5 Übersetzer-Generatoren

Dadurch, dass die Erstellung eines Scanners (lexikalische Analyse) und eines Parser (Syntaxanalyse) eine mechanische Aufgabe ist, bietet sich an, diese zu automatisieren. Eine Software, welche diese Aufgabe automatisiert, wird als Übersetzer-Generator bezeichnet. Übersetzer-Generatoren erfordern meist eine Grammatik-Spezifikation der Quellsprache als Input, welche Symbolspezifikationen (für den Scanner) und Produktionsregeln (für den Parser) umfasst. Darüber hinaus erlauben manche Übersetzer-Generatoren Attributierungen sowie die Einbindung von Quellcode der Implementierungssprache um semantische Aktionen realisieren zu können. [Kas99]

Bekannte Übersetzer-Generatoren für Java sind ANTLR [Par], JavaCC [Mic] oder Coco/R [LMW]. Im Abschnitt 7.3.6 wird der für die Implementierung der Parser-Komponente verwendete Übersetzer-Generator JavaCC näher beschrieben.

6.3 Zusammenfassung

Dieses Kapitel diente der Vermittlung der theoretischen Grundlagen des Übersetzerbaus, um die im Kapitel 7 folgende Dokumentation der Entwicklung der Parser-Komponente für SQL-MDi verstehen und beurteilen zu können.

Zu den theoretischen Grundlagen des Übersetzerbaus ist im weiteren Sinn auch die Theorie der formalen Sprachen und Automaten zu zählen, welche im ersten Teil dieses Kapitels behandelt wurde. Für Programmiersprachen und Abfragesprachen sind vor allem die regulären und kontextfreien Sprachen relevant. Reguläre Sprachen können durch reguläre Grammatiken, reguläre Ausdrücke und endliche Automaten beschrieben werden; kontextfreie Sprachen können durch kontextfreie Grammatiken und Kellerautomaten beschrieben werden.

Der zweite Teil dieses Kapitels behandelte den Übersetzerbau im engeren Sinn, insbesondere die für die Entwicklung der Parser-Komponente relevanten Übersetzungsphasen „lexikalische Analyse“, „Syntaxanalyse“ und „Semantikverarbeitung“.

Im Zuge der lexikalischen Analyse (Scanner) wird eine Input-Zeichenkette in eine Folge von Symbolen zerlegt, wobei bedeutungslose Zeichen und Kommentare überlesen werden. Für die Spezifikation der Symbole werden oft reguläre Ausdrücke verwendet. Die Syntaxanalyse (Parser) prüft die syntaktische Korrektheit einer Input-Zeichenkette, indem festgestellt wird, ob die vom Scanner gelieferten Symbolfolgen mit der (meist kontextfreien) Grammatik der Quellsprache erzeugt werden können. Die Erzeugung der Symbolfolgen erfolgt mit einem Syntaxbaum, welcher top-down (für LL(k) Grammatiken) oder bottom-up (für LR(k) Grammatiken) aufgebaut wird. Werden Syntaxfehler erkannt, wird eine Fehlermeldung erzeugt und optimalerweise die Analyse mit Hilfe von Fangsymbolen fortgesetzt. Aufgabe der Semantikverarbeitung ist die Prüfung kontextabhängiger Eigenschaften und der Aufbau von Datenstrukturen bzw. eines Zwischencodes mit Hilfe von Symboltabellen und attributierten Grammatiken.

Durch die Strukturierung in Phasen kann die Komplexität der Entwicklung eines Übersetzers bzw. der Parser-Komponente für SQL-MDi reduziert werden.

Kapitel 7

SQL-MDi Query Parser

Nachdem in den Kapiteln 5 und 6 die für die Entwicklung einer Parser-Komponente für die multidimensionale Abfragesprache SQL-MDi relevante Theorie erläutert wurde, liegt der Schwerpunkt dieses Kapitels auf deren Realisierung als Software.

Als Bezeichnung für die entwickelte Software wurde „SQL-MDi Query Parser“ gewählt. In den folgenden Ausführungen wird diese Bezeichnung oder die davon abgeleitete Kurzform „Query Parser“ verwendet.

Der erste Abschnitt dieses Kapitels beschreibt die funktionalen und nicht-funktionalen Anforderungen, welche an den SQL-MDi Query Parser gestellt wurden. Abschnitt 7.2 veranschaulicht die Einbettung der Parser-Komponente in das im Abschnitt 3.3.2 vorgestellte Architekturmodell eines Föderierten Data Warehouse-Systems (FDWS). Die bei der Entwicklung des Query Parser verwendeten Technologien und Standards werden im Abschnitt 7.3 beschrieben. Nach der Behandlung dieser Rahmenbedingungen, folgt der Hauptteil dieses Kapitels, die Dokumentation der Architektur und der Implementierung des SQL-MDi Query Parser in den Abschnitten 7.4 und 7.5. Darüber hinaus wird im Abschnitt 7.6 beispielhaft demonstriert, wie Systemänderungen und -erweiterungen vorgenommen werden können. Der letzte Abschnitt 7.7 stellt eine Zusammenfassung des Kapitels dar.

7.1 Anforderungen an den SQL-MDi Query Parser

Dieser Abschnitt dient der Erläuterung der funktionalen und nicht-funktionalen Anforderungen, welche der SQL-MDi Query Parser zu erfüllen hat.

7.1.1 Funktionale Anforderungen

Die folgende Abbildung zeigt ein UML Anwendungsfall-Diagramm [HKKR05], welches die funktionalen Anforderungen übersichtlich darstellt:

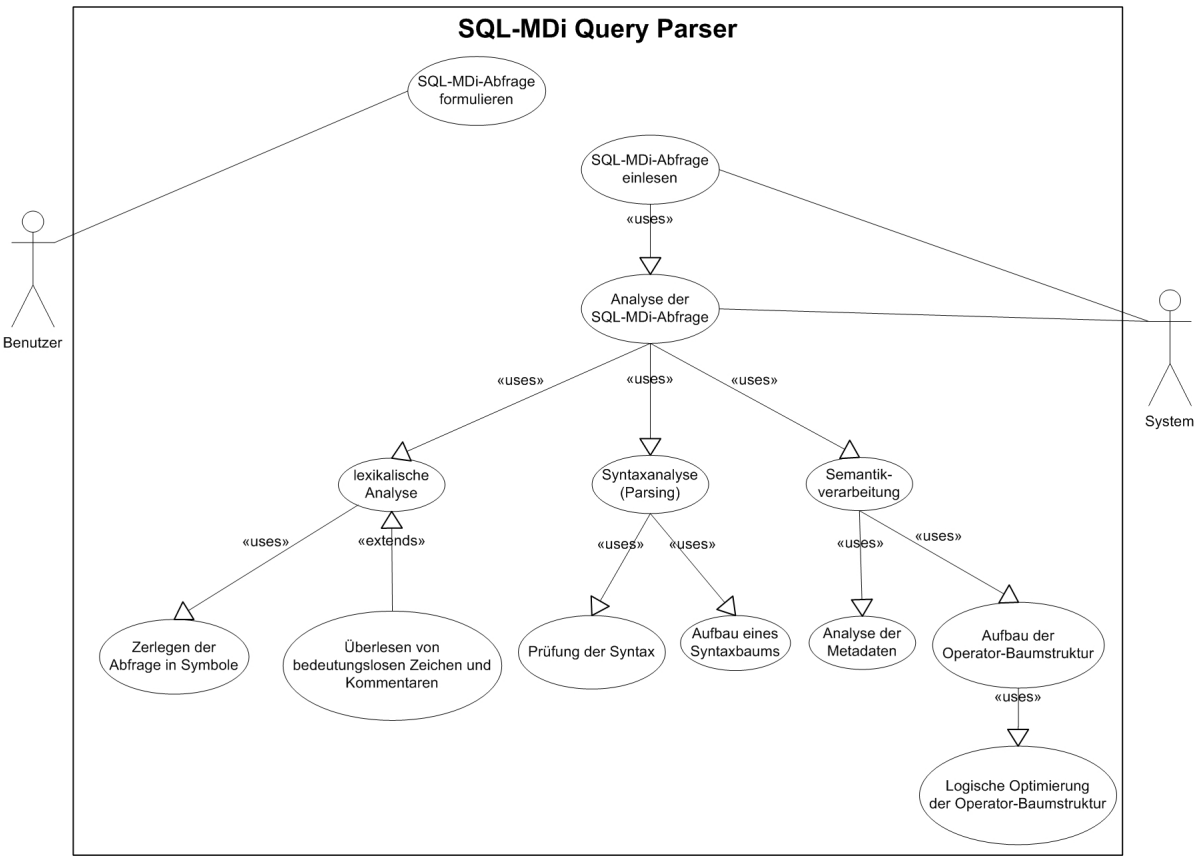


Abbildung 7.1: Anwendungsfall-Diagramm: SQL-MDi Query Parser

Die vom Benutzer formulierte SQL-MDi-Abfrage muss zunächst vom System (= SQL-MDi Query Parser) eingelesen und im Speicher gehalten werden. Ist das geschehen, kann der Query Parser die SQL-MDi-Abfrage analysieren. Diese Analysetätigkeit lässt sich in drei Teilaufgaben gliedern (vgl. Abschnitt 6.2.1):

- Im Zuge der lexikalischen Analyse wird die SQL-MDi-Abfrage in eine Folge von Symbolen zerlegt, wobei hier ggf. bedeutungslose Zeichen (z.B. Leerzeichen, Zeilenumbrüche, etc.) und Kommentare überlesen werden.
- Im Zuge der Syntaxanalyse prüft der Query Parser die Korrektheit der Struktur der SQL-MDi-Abfrage, d.h. die Syntax. Wird ein Syntaxfehler erkannt, erzeugt der Query Parser eine aussagekräftige Fehlermeldung mit Angabe des Symbolwerts und der Symbolposition des fehlerhaften Symbols. Während dieser Analysephase wird auch ein Syntaxbaum (vgl. Abschnitt 6.2.3) erzeugt, welcher die Symbolfolgen strukturiert.
- Im Zuge der Semantikverarbeitung, wird die Korrektheit der in der SQL-MDi-Abfrage enthaltenen Metadaten (z.B. Informationen über Würfel, Dimensionen, Kenngrößen, etc.) durch Abgleich mit einem Metadaten-Repository (vgl. Abschnitt 1.1), welches die Schemainformationen der zu integrierenden DWS verwaltet, überprüft. Darüber hinaus wird eine Operator-Baumstruktur mit den Operatoren der Dimension-Algebra (DA) und

Fact-Algebra (FA) (vgl. Abschnitt 5.3.3) konstruiert.

Im Zuge der Semantikverarbeitung können zwei Arten von Fehlern auftreten: (1) durch die Prüfung der Metadaten mit dem Repository werden Fehler erkannt (z.B. ein deklarierter Würfel existiert im Repository nicht) oder (2) die vom Benutzer formulierte SQL-MDi-Abfrage erzeugt eine Baumstruktur mit logischen Konflikten zwischen Operatoren (vgl. Abschnitt 5.3.4.3). Im ersten Fall wird der Benutzer durch eine Fehlermeldung mit Angabe des fehlerhaften Metadatums und der Fehlerposition informiert. Im zweiten Fall wird eine Fehlermeldung erzeugt, welche Informationen über den konfliktären Operator beinhaltet und Hinweise auf die Konfliktursache bzw. auf eine mögliche Konfliktbereinigung enthält.

Für die drei Analysephasen können demnach, unter Bezugnahme auf die theoretischen Grundlagen des Abschnitts 6.2, die funktionalen Anforderungen wie folgt zusammengefasst werden:

1. **Lexikalische Analyse (vgl. Abschnitt 6.2.2)**
   a) Zerlegung der SQL-MDi-Abfrage in eine Folge von Symbolen, wobei ein numerischer Symbolcode, der Symbolwert und die Symbolposition erfasst werden.

      Der Symbolcode ist eine numerische Konstante, welche vom Parser für die Referenzierung eines Symbols verwendet wird; der Symbolwert enthält die dem Symbol zugeordnete Zeichenkette (z.B. Schlüsselwort „CUBE“); die Symbolposition beschreibt die Position des Symbols in der SQL-MDi-Abfrage durch Angabe der Zeilen- und Spaltennummer.
   b) Ignorieren bedeutungsloser Zeichen
      Vom Benutzer eingegebene Leerzeichen, Zeilenumbrüche, Tabulatoren, diverse Sonderzeichen sowie Umlaute werden vom Scanner ignoriert.
   c) Definition von Kommentaren
      Der Benutzer soll die Möglichkeit haben, Anweisungen in einer SQL-MDi-Abfrage zu kommentieren. Diese Kommentare müssen daher vom Parser ignoriert werden, d.h. deren Syntax wird nicht überprüft.
   d) Fehlerfall „Symbol wird nicht erkannt“: Generierung einer aussagekräftigen Fehlermeldung mit Angabe des nicht erkannten Symbols sowie dessen Position.
2. **Syntaxanalyse (vgl. Abschnitt 6.2.3)**
   a) Prüfung der syntaktischen Korrektheit der SQL-MDi-Abfrage, indem die Übereinstimmung der Symbolfolgen mit den Produktionsregeln der SQL-MDi-Grammatik festgestellt wird.
   b) Aufbau eines Syntaxbaums, welcher die für die Konstruktion der Operator-Baumstruktur erforderlichen Daten (= Symbole) strukturiert.
   c) Fehlerfall „Syntaxfehler“: Generierung einer aussagekräftigen Fehlermeldung mit Angabe des fehlerhaften Symbols sowie dessen Position.

3. **Semantikverarbeitung (vgl. Abschnitt 6.2.4)**
   a) Prüfung der Metadaten
      i. Prüfung der Korrektheit der in der SQL-MDi-Abfrage enthaltenen Metadaten durch Abgleich mit dem Metadaten-Repository (vgl. Abschnitt 1.1).
      ii. Fehlerfall „Ungültiges Metadatum“: Generierung einer aussagekräftigen Fehlermeldung mit Angabe des fehlerhaften Metadatums sowie dessen Position.
   b) Aufbau der Operator-Baumstruktur mit den DA/FA-Operatoren
      i. Die Operator-Baumstruktur ist als Ausführungsplan für die Prozessor-Komponente zu definieren, d.h. die Berechnung eines integrierten Ergebnisses aufgrund der Baumstruktur muss, ohne Umstrukturierungen vorzunehmen, möglich sein.
      ii. Berücksichtigung von Optimierungsmöglichkeiten auf logischer Ebene um eine effiziente Abfrageverarbeitung zu unterstützen.
      iii. Fehlerfall „logischer Konflikt zwischen Operatoren“: Generierung einer aussagekräftigen Fehlermeldung mit Angabe des konfliktären Operators sowie eines Hinweises auf die Konfliktursache bzw. auf eine Möglichkeit zur Konfliktbereinigung.

7.1.2 Nicht-funktionale Anforderungen

In diesem Abschnitt werden die nicht-funktionalen Anforderungen an den SQL-MDi Query Parser erläutert:

1. **Änderbarkeit**
   Änderungen in der SQL-MDi-Syntax bzw. in der DA/FA müssen möglichst einfach im Query Parser nachgezogen werden können, d.h. ohne große Änderungen an bestehendem Quellcode vornehmen zu müssen.
2. **Benutzerinteraktion**
   Aufgrund des gewählten prototypischen Ansatzes (vgl. „6. Prototypischer Ansatz“) muss die entwickelte Version des SQL-MDi Query Parser keine graphische Benutzerschnittstelle umfassen. Die Interaktion mit dem Benutzer kann sich auf einen „File-Dialog“ beschränken, mit welchem der Benutzer eine Text-Datei mit einer SQL-MDi-Abfrage selektieren und dem Query Parser zur Verarbeitung übertragen kann.
3. **Erweiterbarkeit**
   Erweiterungen der SQL-MDi-Syntax um neue Anweisungen bzw. der DA/FA um neue Operatoren müssen möglichst einfach im Query Parser implementiert werden können, d.h. ohne Änderungen an bestehendem Quellcode vornehmen zu müssen.
4. **Interoperabilität**
   Der SQL-MDi Query Parser muss unabhängig von den zu integrierenden Komponenten-DWS realisiert werden. Dafür ist ein zentrales Repository zur Speicherung der Schemainformationen der Komponenten-DWS vorgesehen. Anstatt die in einer SQL-MDi-Abfrage enthaltenen Metadaten direkt mit den Schemainformationen aus den Komponenten-DWS abzugleichen, hat der Abgleich mit dem Repository zu erfolgen. Daher kann der

Query Parser Schemainformationen beliebiger Komponentensysteme handhaben, solange diese vom Repository verwaltet werden.

5. **Plattformunabhängigkeit**
   Der SQL-MDi Query Parser muss auf Windows-, Linux- und Unix-Plattformen lauffähig sein.
6. **Prototypischer Ansatz**
   Die Entwicklung des SQL-MDi Query Parser hat einem prototypischen Ansatz zu folgen, mit welchem die entwickelten theoretischen Konzepte evaluiert werden können (vgl. Kapitel 5). Für die Evaluierung der theoretischen Konzepte werden relationale Star-Schemata (vgl. Abschnitt 2.4.2.1) als logische Modelle der Komponenten-DWS angenommen, d.h. dass beispielsweise die Einsetzbarkeit für MOLAP-Systeme (vgl. Abschnitt 2.4.2) nicht geprüft werden muss. Aus diesen Gründen ist die Einsetzbarkeit des Query Parser in einer Produktivumgebung explizit nicht Ziel dieser Diplomarbeit.
7. **Robustheit**
   Wie bereits im Abschnitt 7.1.1 bei den funktionalen Anforderungen erwähnt, muss der SQL-MDi Query Parser im Fehlerfall aussagekräftige Meldungen zur Information des Benutzers generieren. Um dem Benutzer das Formulieren einer korrekten SQL-MDi-Abfrage zu erleichtern, darf der Query Parser die Verarbeitung beim Auftreten des ersten Fehlers nicht abbrechen, sondern soll einen möglichst großen Teil der Abfrage analysieren und am Ende alle Fehlermeldungen gesammelt ausgeben, d.h. robust sein. Diese Anforderung wurde bereits in den theoretischen Grundlagen des Übersetzerbaus (vgl. Abschnitt 6.2.3.3) erläutert.
8. **Wiederverwendbarkeit**
   Der SQL-MDi Query Parser muss unabhängig von der den Output (Operator-Baumstruktur) verarbeitenden Prozessor-Komponente realisiert werden, sodass die Baumstruktur ggf. für andere Zwecke verwendet werden kann.

Nachdem in diesem Abschnitt die funktionalen und nicht-funktionalen Anforderungen an den SQL-MDi Query Parser erläutert wurden, wird im folgenden Abschnitt seine Integration in die im Abschnitt 3.3.2 vorgestellte Architektur eines FDWS dargestellt.

7.2 Integration in FDWS-Architektur

Abbildung 7.2 zeigt die Software-Komponenten der Föderations-Schicht des im Abschnitt 3.3.2 präsentierten Architekturmodells eines FDWS, nämlich ein Integrations-Werkzeug und ein Abfrage-Werkzeug, bestehend aus dem SQL-MDi Query Parser und einem Query Prozessor. Die Trennung von Query Parser und Query Prozessor unterstützt die „Wiederverwendbarkeit“ des Query Parser (vgl. nicht-funktionale Anforderung 8 im Abschnitt 7.1.2). Die strichlierten Pfeile stellen Metadatenflüsse dar; Pfeile mit durchgezogenen Linien repräsentieren Datenflüsse im Zuge der Verarbeitung einer SQL-MDi-Abfrage.

Ein wesentliches Charakteristikum dieses Architekturmodells ist die Verfügbarkeit eines „Metadaten-Repository“, welches der zentralen Speicherung der Schemainformationen der Komponenten-DWS bzw. des FDWS dient. Architekturen mit einem zentralen Metadaten-Repository werden auch als „hub-and-spoke Metadaten-Architekturen“ bezeichnet [PCTM02]. Die Verfügbarkeit des zentralen Metadaten-Repository gewährleistet die „Interoperabilität“ des Query Parser mit beliebigen Komponentensystemen (vgl. nicht-funktionale Anforderung 4 im Abschnitt 7.1.2).

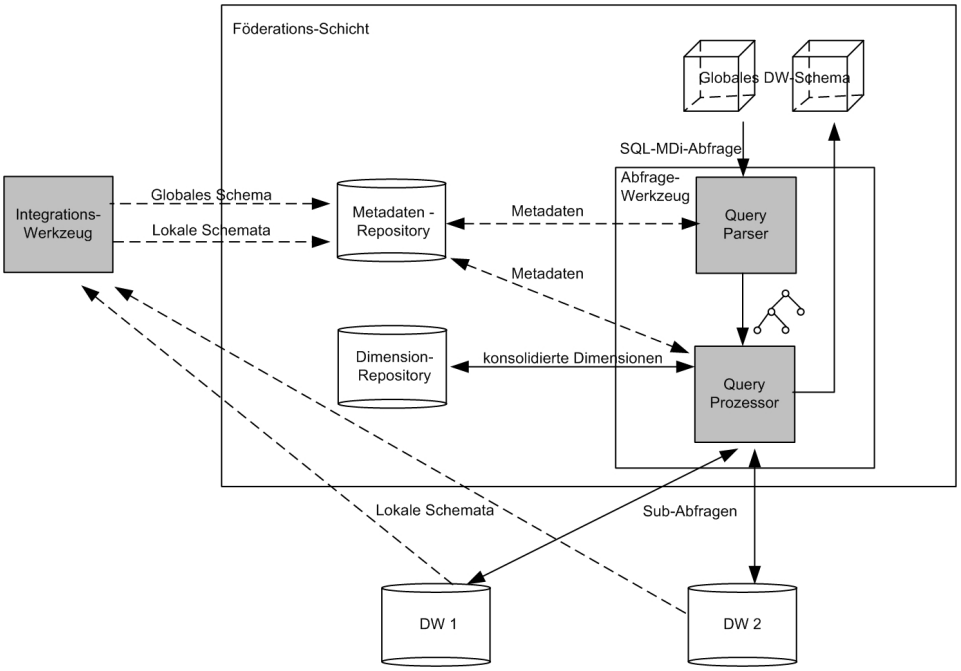


Abbildung 7.2: Software-Komponenten im FDWS

7.2.1 Integrations-Werkzeug

Die Konzeption und Entwicklung der Software-Komponente „Integrations-Werkzeug“ ist Gegenstand einer weiteren, zur Zeit in Arbeit befindlichen Diplomarbeit [Sch08]. Das Integrations-Werkzeug unterstützt den FDWS-Administrator bei der Erstellung eines integrierten, globalen DW-Schemas aus den Schemata der Komponenten-DWS. Mit diesem Werkzeug kann dieser die Schemata der lokalen DWS extrahieren und interaktiv ein globales DW-Schema definieren. Die lokalen Schemata und das definierte globale Schema werden in einer Datenbank, dem Metadaten-Repository, nach dem CWM(Common Warehouse Metamodel)-Standard gespeichert [OMG01] (vgl. Abschnitt 7.3.8).

7.2.2 Query Parser

Die Aufgabe der Software-Komponente „Query Parser“ ist die Transformation einer SQL-MDi-Abfrage in eine Operator-Baumstruktur mit den DA/FA-Operatoren (vgl. Abschnitt 5.3.3). Eine ausführliche Erläuterung der Architektur und der Implementierung des Query Parser folgt in den Abschnitten 7.4 und 7.5. Zur Prüfung der in einer SQL-MDi-Abfrage enthaltenen Metadaten, z.B. Informationen über Kenngrößen, Dimensionen, Klassifikationsstufen, etc., greift der Query Parser auf die Schemainformationen des Metadaten-Repository zu.

7.2.3 Query Prozessor

Die Konzeption und Entwicklung der Software-Komponente „Query Prozessor“ ist Gegenstand einer weiteren, zur Zeit in Arbeit befindlichen Diplomarbeit [Ros08]. Die erste Teilaufgabe des Query Prozessors ist die Transformation der vom Query Parser generierten Operator-Baumstruktur in eine Menge von Sub-Anweisungen für die Komponenten-DWS, welche in einem Ausführungsplan angeordnet werden. Die zweite Teilaufgabe ist die Abarbeitung dieses Ausführungsplans, wodurch das integrierte Ergebnis der SQL-MDi-Abfrage, das globale, instanziierte DW-Schema bzw. der globale, instanziierte Würfel, berechnet wird.

Bevor die Architektur und die Implementierung des SQL-MDi Query Parser erläutert werden, werden im nächsten Abschnitt die für die Entwicklung verwendeten Technologien und Standards beschrieben.

7.3 Technologien und Standards

In diesem Abschnitt werden die für die Entwicklung des SQL-MDi Query Parser verwendeten Technologien, wie die Programmiersprache, Programmbibliotheken, Entwicklungsumgebung und sonstige Werkzeuge kurz vorgestellt. Darüber hinaus erfolgt ein kurzer Überblick über den bereits erwähnten CWM-Standard [OMG01].

7.3.1 Programmiersprache Java 6.0

Die Wahl der verwendeten Programmiersprache hängt von drei wesentlichen Kriterien ab:

- Verträglichkeit mit den anderen Software-Komponenten des FDWS (vgl. Abschnitt 7.2), insbesondere mit dem Metadaten-Repository und der Prozessor-Komponente
- Unterstützung einer plattformunabhängigen Implementierung (vgl. nicht-funktionale Anforderung 5 im Abschnitt 7.1.2)

- Verfügbarkeit von Bibliotheken und Werkzeugen für eine Programmiersprache zur Unterstützung der Implementierung um die funktionalen und nicht-funktionalen Anforderungen (vgl. Abschnitt 7.1) umfassend erfüllen zu können.

Da sich für die Entwicklung des Integrations-Werkzeugs Java [Javc] als die zweckmäßigste Programmiersprache erwies [Sch08], wurde diese im Sinne eines homogenen Gesamtsystems auch für die Entwicklung des Abfrage-Werkzeugs, d.h. auch des Query Parser, gewählt. Darüber hinaus gewährleistet Java die Erfüllung der nicht-funktionalen Anforderungen „Plattformunabhängigkeit“ sowie aufgrund der Objektorientierung „Änderbarkeit“ und „Erweiterbarkeit“ (vgl. nicht-funktionale Anforderungen 5, 1 und 3 im Abschnitt 7.1.2). Die Verträglichkeit mit dem Metadaten-Repository kann mit der JDBC API (vgl. Abschnitt 7.3.2) sichergestellt werden. Weiters existiert eine Reihe von Java-basierten Werkzeugen für die Parser-Implementierung (vgl. Abschnitt 6.2.5) um die funktionalen und nicht-funktionalen Anforderungen umfassend zu erfüllen.

7.3.2 Java Database Connectivity (JDBC)

Die JDBC API ist ein Standard für datenbankunabhängige Verbindungen zwischen Java-Programmen und einer großen Auswahl von Datenbanken. Mit JDBC kann der Vorteil von Java „Write Once, Run Anywhere“ genutzt werden um datenbankgestützte Anwendungen zu programmieren. Diese API bietet Klassen und Methoden für den Verbindungsaufbau zu einer Datenbank, für das Senden von SQL-Anweisungen und das Verarbeiten der returnierten Ergebnisse. [Java]

Im Query Parser-System wird diese API für den Zugriff auf die Datenbank des Metadaten-Repository verwendet.

7.3.3 Log4j 1.2.14

Log4j ist eine Open Source-Logging API für Java, mit welcher Log-Anweisungen in den Quellcode eingebunden werden können. Log4j ist mit Konfigurationsdateien flexibel konfigurierbar, indem beispielsweise für Log-Anweisungen unterschiedliche Level (z.B. FATAL) definiert und die Anweisungen abhängig vom Level in unterschiedliche Log-Dateien geschrieben werden können. Die Erzeugung von Log-Dateien bzw. die Ausgabe von Log-Anweisungen unterstützt das Debugging von Anwendungen, da der Programmablauf damit dokumentiert werden kann. [Log]

Im Query Parser-System wird diese API für die Dokumentation der Erzeugung von Metadaten-Objekten (z.B. für Würfel, Dimensionen, Kenngrößen, etc.) und des Aufbaus der Operator-Baumstruktur verwendet.

7.3.4 JUnit 4.0

JUnit ist eine Open Source-Test API für Java, welche die Implementierung von Unit Tests in Java unterstützt. Ein Unit-Test dient der isolierten Prüfung einer ausführbaren Programmeinheit (z.B. einer Methode oder Klasse) [ZR06]. Mit dieser Bibliothek können Black-Box Tests [ZR06] für Methoden implementiert werden, indem mit Hilfe spezieller Testmethoden geprüft wird, ob das tatsächliche Ergebnis eines Methodenaufrufs dem erwarteten Ergebnis entspricht. [JUn]

Diese API wurde für die Erstellung von Unit Tests für die Sicherstellung einer fehlerfreien Benutzung des Query Parser verwendet.

7.3.5 Eclipse 3.2

Eclipse wurde als Java-Entwicklungsumgebung für die Implementierung des Query Parser verwendet. Darüber hinaus unterstützt Eclipse auch die Entwicklung von Rich Client-Anwendungen, indem die bereitgestellte Kernfunktionalität in Form von Plugins erweitert werden kann. [Ecl]

Ein Grund für die Verwendung von Eclipse als Entwicklungsumgebung war die Verfügbarkeit eines Plugins für den Übersetzer-Generator „JavaCC“ (vgl. Abschnitt 7.3.7).

7.3.6 JavaCC 4.0

Da die Implementierung eines Parser eine mechanische, reproduzierbare Aufgabe ist, wurden diverse Software-Werkzeuge entwickelt um diese zu automatisieren. Derartige Software-Werkzeuge, welche die automatische Erzeugung eines Scanners und eines Parser aus einer Grammatik-Spezifikation der Quellsprache ermöglichen, werden als Übersetzer-Generatoren bezeichnet (vgl. Abschnitt 6.2.5).

JavaCC ist der populärste Übersetzer-Generator für Java-Anwendungen, welcher aus einer Grammatik-Spezifikation ein Java-Programm generiert, welches gültige Sätze dieser Grammatik erkennt. [Mic]

Da der Aufwand für die Implementierung des Query Parser durch Verwendung eines geeigneten Übersetzer-Generators wesentlich reduziert werden kann, wurden die gebräuchlichsten Werkzeuge hinsichtlich ihrer Eignung für die Themenstellung analysiert. Generatoren, welche nur die automatische Erzeugung entweder eines Scanners oder eines Parser unterstützen, wurden nicht näher betrachtet. Folgende frei verfügbare Übersetzer-Generatoren für Java wurden untersucht: ANTLR [Par], Coco/R [LMW] und JavaCC [Mic].

Analyse von Übersetzer-Generatoren

Im Folgenden werden die Kriterien für die Beurteilung der Übersetzer-Generatoren hinsichtlich ihrer Eignung für die Implementierung des SQL-MDi Query Parser vorgestellt, wobei zu den jeweiligen Kriterien angegeben wird, ob sie den generierten Scanner oder Parser betreffen. Die Kriterien wurden großteils aus den funktionalen und nicht-funktionalen Anforderungen des Abschnitts 7.1 abgeleitet.

1. **Definition von Kommentaren** (Scanner)
   Der Benutzer soll die Möglichkeit haben, Anweisungen einer SQL-MDi-Abfrage zu kommentieren. Der Übersetzer-Generator hat daher die Deklaration spezieller Symbole für Kommentare zu unterstützen, welche vom Parser überlesen werden. (vgl. funktionale Anforderung 1.c im Abschnitt 7.1.1)
2. **Berücksichtigen/Ignorieren von Groß- und Kleinschreibung** (Scanner)
   Der Übersetzer-Generator muss eine Option bieten, aufgrund welcher ein Scanner erzeugt wird, der Groß- und Kleinschreibung in einer SQL-MDi-Abfrage berücksichtigt bzw. ignoriert.
3. **Spezifikation von zu ignorierenden Zeichen** (Scanner)
   Vom Benutzer eingegebene Leerzeichen, Zeilenumbrüche, Tabulatorzeichen, etc. müssen ignoriert werden können. (vgl. funktionale Anforderung 1.b im Abschnitt 7.1.1)
4. **Flexible Spezifikation der Anzahl der Vorgriffssymbole**
   (vgl. Abschnitt 6.2.3) (Parser)
   Da die SQL-MDi-Grammatik teilweise nicht LL(1) bzw. LR(1) ist, d.h. die richtige Alternative im Falle einer Verzweigung mit einem Vorgriffssymbol nicht festgestellt werden kann, muss der Übersetzer-Generator die flexible Erhöhung der Anzahl der Vorgriffssymbole unterstützen. Eine solche Situation tritt auf, wenn zwei oder mehrere Folgeregeln mit demselben Symbol, z.b. '(', beginnen.
5. **Unterstützung der Semantikverarbeitung** (Parser)
   Der Query Parser muss semantische Aktionen zur Prüfung der Metadaten und zum Aufbau der Operator-Baumstruktur ausführen können. (vgl. funktionale Anforderungen 3.a und 3.b im Abschnitt 7.1.1)
   a) Einbindung von Java-Code
      Ein Übersetzer-Generator muss die Einbindung von beliebigem Java-Code als semantische Aktionen unterstützen um z.B. Datenstrukturen aufzubauen.
   b) Unterstützung von Attributierungen (vgl. Abschnitt 6.2.4.2)
      Um semantische Aktionen, welche mehrere Produktionsregeln betreffen, realisieren zu können, muss ein Datenaustausch zwischen Produktionsregeln mittels Attributierungen möglich sein.
   c) Trennung von Grammatik-Spezifikation und Semantikverarbeitung
      Da der Query Parser eine Vielzahl semantischer Aktionen ausführen muss, sollte deren Einbindung getrennt von der Grammatik-Spezifikation erfolgen können um die Änderbarkeit und Erweiterbarkeit zu erhöhen (vgl. nicht-funktionale Anforderungen 1 und 3 im Abschnitt 7.1.2)

6. **Fehlerbehandlung** (Parser)
   a) Aussagekräftige Fehlermeldungen
   Der vom Übersetzer-Generator erzeugte Parser muss beim Erkennen eines Syntaxfehlers bereits standardmäßig aussagekräftige Fehlermeldungen mit Angabe des fehlerhaften Symbols sowie dessen Position in der SQL-MDi-Abfrage erzeugen. (vgl. funktionale Anforderung 2.c im Abschnitt 7.1.1)
   b) Unterstützung des Wiederaufsatzes (vgl. Abschnitt 6.2.3.3)
   Der Übersetzer-Generator muss Unterstützung für den automatischen Wiederaufsatz zur Fortsetzung des Parsing im Falle von Syntaxfehlern bieten. (vgl. nichtfunktionale Anforderung 7 im Abschnitt 7.1.2)
7. **Verfügbarkeit eines Entwicklungswerkzeugs** (Scanner und Parser) („nice-to-have“)
   Wünschenswert wäre die Verfügbarkeit eines Entwicklungswerkzeugs, welches die Parser-Entwicklung mit dem Übersetzer-Generator erleichtert.

Tabelle 7.1 zeigt eine Zusammenfassung des Ergebnisses der Analyse der genannten Übersetzer-Generatoren.

Kriterium	**ANTLR**	**Coco/R**	**JavaCC**
1. Definition von Kommentaren	✓	✓	✓
2. Groß-/Kleinschreibung		✓	✓
3. Zeichen ignorieren	✓	✓	✓
4. Anzahl Vorgriffssymbole	✓	✓	✓
5. Unterstützung Semantikverarbeitung			
a) Einbindung von Java-Code	✓	✓	✓
b) Attributierungen	✓	✓	✓
c) Trennung Grammatik-Spezifikation/Semantikverarbeitung	✓		✓
6. Fehlerbehandlung			
a) Aussagekräftige Fehlermeldungen	✓	✓	✓
b) Unterstützung des automatischen Wiederaufsatzes	✓	✓	✓
7. Entwicklungswerkzeug	(✓)	(✓)	(✓)

Tabelle 7.1: Beurteilung der Übersetzer-Generatoren

Wie aus dem Analyseergebnis ersichtlich, sind die untersuchten Übersetzer-Generatoren hinsichtlich der Erfüllung der relevanten Kriterien als nahezu gleichwertig zu beurteilen.

Jedoch bietet ANTLR, soweit bekannt, keine Option für das Berücksichtigen/Ignorieren der Groß- und Kleinschreibung. Soll daher ein Scanner generiert werden, welcher Groß- und Kleinschreibung ignoriert, so muss dies bei der Spezifikation der Symbole vorgesehen werden. Eine nachträgliche Änderung auf „Berücksichtigung der Groß- und Kleinschreibung“ ist daher relativ umständlich.

Alle drei untersuchten Übersetzer-Generatoren ermöglichen die Einbindung von Java-Quellcode als semantische Aktionen und von Attributierungen in die Input-Grammatik-Spezifikation zur

Unterstützung der Semantikverarbeitung. ANTLR und JavaCC unterstützen alternativ die Erzeugung eines Syntaxbaums („Abstract Syntax Tree“ (AST)) aus einer SQL-MDi-Abfrage, wobei in den AST-Knoten die für die Semantikverarbeitung erforderlichen Daten gekapselt werden können. Im Zuge der Traversierung des AST können semantische Aktionen für die AST-Knoten ausgeführt werden, wodurch eine Trennung von Grammatik-Spezifikation und Semantikverarbeitung ermöglicht wird. Coco/R bietet keine Unterstützung zur Trennung von Grammatik-Spezifikation und Semantikverarbeitung, weshalb die Änderbarkeit und Erweiterbarkeit des Query Parser negativ beeinträchtigt wird. Es besteht natürlich die Möglichkeit den Aufbau eines AST im erzeugten Parser manuell zu implementieren, was jedoch bei einer komplexen Grammatik relativ aufwendig sein kann.

Da JavaCC als einziger Übersetzer-Generator alle genannten Kriterien erfüllt und da dessen Funktionalität darüber hinaus sehr gut dokumentiert ist, wurde dieser für die Entwicklung des SQL-MDi Query Parser verwendet.

Die nun folgenden Erläuterungen sollen einen kurzen Überblick über die von JavaCC unterstützten Funktionalitäten geben. Beispiele für die Anwendung von JavaCC werden im Zuge der Beschreibung der SQL-MDi Query Parser-Implementierung im Abschnitt 7.5 präsentiert.

Funktionalität

Die folgende Aufzählung dokumentiert die für die Entwicklung des Query Parser wichtigsten Funktionalitäten von JavaCC: [Mic]

- JavaCC erzeugt einen Parser, welcher einen Syntaxbaum von oben nach unten generiert (= top-down Parser), d.h. JavaCC kann LL(k) Grammatiken verarbeiten (vgl. Abschnitt 6.2.3.1). Standardmäßig generiert JavaCC einen LL(1)-Parser, wobei jedoch für nicht LL(1)-Teile der Grammatik die Anzahl der Vorgriffsymbole erhöht werden kann.
- JavaCC ermöglicht den Aufbau eines AST, wodurch eine Trennung von Grammatik-Spezifikation und Semantikverarbeitung ermöglicht wird.
- JavaCC zeichnet sich durch die Aussagekraft und Anpassbarkeit der Fehlermeldungen aus, indem standardmäßig bereits die genaue Fehlerposition ausgegeben wird und die Fehlermeldungen ohne großen Aufwand an die eigenen Bedürfnisse angepasst werden können.
- JavaCC stellt weitere nützliche Funktionalitäten, z.B. für die Dokumentation der Grammatik-Spezifikation, für die Spezifikation von Kommentaren und das Debugging, zur Verfügung.

JavaCC-Grammatik-Spezifikation

Im Folgenden wird ein kurzer Überblick über den Aufbau einer JavaCC-Grammatik-Spezifikation gegeben:

Zu Beginn können diverse globale Variablen (z.B. Anzahl der Vorgriffssymbole, Debugging-Optionen, Berücksichtigen/Ignorieren der Groß- und Kleinschreibung, etc.) definiert werden. Im Anschluss folgt ein Java-Code-Programm zwischen den Anweisungen `BEGIN PARSER(name)` und `END PARSER(name)`, welcher die Übersetzungseinheit repräsentiert. Dort wird u.a. das Einlesen des zu parsenden Inputs und das Starten des Parsing programmiert. Nach der Übersetzungseinheit folgt eine Menge von Produktionsregeln, wobei folgende Typen unterschieden werden können: [Mic]

- javacode_production
- regular_expr_production
- bnf_production
- token_manager_decls

javacode_production und bnf_production werden für die Definition der Produktionsregeln der Grammatik verwendet, für welche der Parser erstellt werden soll. regular_expr_production und token_manager_decls dienen der Definition von Symbolen (Token), z.B. für Schlüsselwörter und Bezeichner, aus welchen ein so genannter „Token-Manager“ generiert wird. Beispiele für die unterschiedlichen Typen von Produktionsregeln werden im Zuge der Beschreibung der SQL-MDi Query Parser-Implementierung im Abschnitt 7.5 präsentiert.

Wird eine syntaktisch korrekte Grammatik-Spezifikation mit dem JavaCC-Compiler übersetzt, werden folgende Dateien erstellt, wobei der Prefix der Dateinamen vom Namen der Übersetzungseinheit bestimmt wird, z.B. `name` = QueryParser:

- QueryParser.java: der generierte Parser
- QueryParserTokenManager.java: der generierte Token-Manager (oder Scanner)
- QueryParserConstants.java: Konstanten für definierte Symbole
- Darüber hinaus werden diverse Grammatik-unabhängige Dateien erzeugt, z.B.:
  Token.java: repräsentiert Symbole
  ParseException.java: zur Erzeugung von Fehlermeldungen bei Syntaxfehlern

AST Support

JavaCC inkludiert den so genannten JJTree-Präprozessor [Mic], welcher Anweisungen zur Erzeugung von AST-Knoten in die Produktionsregeln des Parser einfügt. Dafür müssen Knoten-Dekorationen zu Produktionsregeln der Grammatik-Spezifikation angegeben werden. JJTree unterstützt die automatische Generierung der entsprechenden AST-Knoten-Klassen, welche jedoch auch manuell erstellt werden können. [Mic]

Für die Verarbeitung des AST besteht die Möglichkeit der automatischen Berücksichtigung des Visitor-Design Patterns [GHJV95] (vgl. Abschnitt 7.5.4.1) in der Implementierung, wodurch für jede AST-Knoten-Klasse eine separate `visit()` Methode zur Verfügung steht, in

welcher semantische Aktionen definiert werden können. [Mic] Beispiele für die Verwendung von JJTree zum Aufbau eines AST werden im Zuge der Beschreibung der SQL-MDi Query Parser-Implementierung im Abschnitt 7.5 präsentiert.

Fehlerbehandlung und Wiederaufsatz

Fehlermeldungen beim Erkennen von Syntaxfehlern werden durch die Klasse `ParseException` repräsentiert, wobei eine Fehlermeldung neben dem fehlerhaften Symbol standardmäßig dessen Position sowie das/die erwartete(n) Symbol(e) umfasst. Um Fehlermeldungen den eigenen Bedürfnissen anzupassen, reicht es meist, die Methode `getMessage()` entsprechend zu verändern, indem die zu erzeugende Ausgabe modifiziert wird. [Mic]

Eine wichtige Anforderung an die Benutzerfreundlichkeit eines Parser ist der automatische Wiederaufsatz, d.h. dass das Parsing beim Erkennen eines Syntaxfehlers nicht abgebrochen werden soll (vgl. Abschnitt 6.2.3.3). Um den automatischen Wiederaufsatz zu ermöglichen, muss die beim Erkennen eines Syntaxfehlers generierte `ParseException` abgefangen und der Parser manuell bis zum nächsten, als gültig angenommenen Zustand gesteuert werden. [Mic] Beispiele für die Anpassung der Fehlerausgabe der Methode `getMessage()` und den automatischen Wiederaufsatz werden im Zuge der Beschreibung der SQL-MDi Query Parser-Implementierung im Abschnitt 7.5 präsentiert.

7.3.7 JavaCC Eclipse Plugin 1.5.7

Dieses Plugin wird als Unterstützung für die Erstellung der JavaCC-Grammatik-Spezifikation verwendet [Javb]. Es bietet u.a. folgende Funktionalitäten:

- User Interface für die Definition von globalen Variablen
- Editor mit Syntax-Highlighting
- Outline der Dokumentstruktur
- Konsolenausgabe von Fehlern und Warnungen
- JavaCC-, JJTree-, JJDoc-Übersetzung
- Kennzeichnung generierter Dateien
- etc.

7.3.8 Common Warehouse Metamodel (CWM)

CWM ist ein Standard, welcher ein gemeinsames Metamodell und XML-basiertes Austauschformat für Metadaten auf dem Gebiet des Data Warehousing definiert. [OMG01]

Allgemeines

Ein gemeinsames Metamodell und Austauschformat unterstützt die Interoperabilität von Data Warehouse-Lösungen unterschiedlicher Hersteller.

Ohne CWM als gemeinsames Metamodell und Austauschformat für Metadaten müssten eine Vielzahl so genannter „Bridges“ für die unterschiedlichen Produkte implementiert werden um einen Austausch von Metadaten zu ermöglichen. Eine Bridge ist eine Software, die Metadaten eines Produkts in ein Format übersetzt, welches von einem anderem Produkt gefordert wird. Existiert in einem solchen Fall kein zentrales Metadaten-Repository, müssen schlimmstenfalls für alle Produktpaare Bridges implementiert werden („point-to-point Metadaten-Architektur“); existiert ein Metadaten-Repository, muss zumindest noch für jedes Produkt eine Bridge für den Metadaten-Austausch mit dem Repository implementiert werden („hub-and-spoke Metadaten-Architektur“).

Mit der Verwendung von CWM entfällt der Aufwand für die Implementierung der Bridges: In point-to-point Architekturen müssen keine Bridges zwischen zu integrierenden Produkt-Typen implementiert werden. Stattdessen implementiert jedes Produkt einmalig einen Adapter, welcher CWM sowie das produktspezifische Metamodell versteht. In hub-and-spoke Architekturen repräsentiert das Repository einen zentralen Speicher für die CWM-Metamodelle und deren Instanzen (Modelle). Die einzelnen Produkte implementieren wiederum einen entsprechenden Adapter. [PCTM02]

CWM-Architektur

Der CWM-Standard ist in 21 Pakete, welche auf fünf Schichten, nämlich „Object Model“, „Foundation“, „Resource“, „Analysis“ und „Management“, verteilt sind, organisiert. Jedes Paket enthält Klassen, Assoziationen und Constraints, welche einem bestimmten Teilgebiet des Data Warehousing zugeordnet sind, z.B. dient das Paket „Relational“ der Modellierung von relationalen Metadaten, das Paket „OLAP“ der Modellierung von multidimensionalen Metadaten. Um ein bestimmtes CWM-Paket zu implementieren, reicht es nicht aus, nur dieses Paket sondern auch jene Pakete, auf welchen das zu implementierende Paket aufbaut, zu verstehen. [PCTM02]

Für die Modellierung des Metadaten-Repository wurden die Pakete „OLAP“, „Transformation“, „Relational“, „Software Deployment“, „Keys and Indizes“, „Core“ und „Relationsships“ verwendet. [Sch08]

7.4 Systemarchitektur

In diesem Abschnitt folgt die Erläuterung der Systemarchitektur des SQL-MDi Query Parser (vgl. Abbildung 7.3), wobei die wichtigsten Software-Komponenten kurz erläutert werden, bevor deren Implementierung im Abschnitt 7.5 im Detail beschrieben wird.

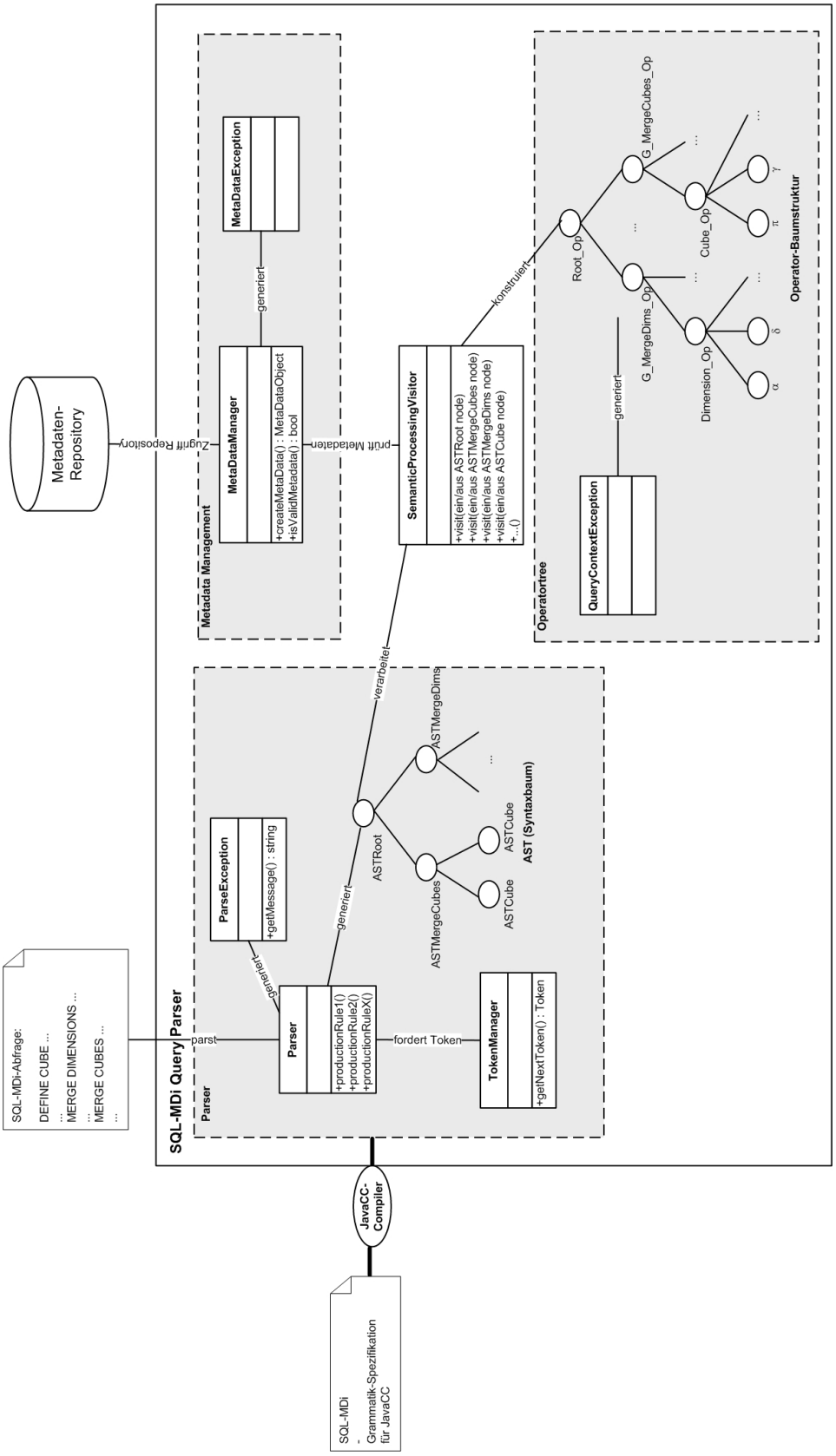


Abbildung 7.3: Systemarchitektur des SQL-MDi Query Parser

Abbildung 7.3 zeigt die Systemarchitektur des SQL-MDi Query Parser, welche aus den drei Komponenten „Parser“, „Metadata Management“ und „Operatortree“ besteht. Die Parser-Komponente mit den Klassen `Parser`, `TokenManager` und `ParseException` sowie den Klassen für die AST-Knoten werden vom JavaCC-Compiler aus einer Grammatik-Spezifikation für SQL-MDi automatisch generiert. Die AST-Knoten kapseln die für die Erzeugung eines DA/FA-Operators erforderlichen Daten. Die Klasse `SemanticProcessingVisitor` dient der Verarbeitung des im Zuge des Parsing generierten AST. Jede `visit()` Methode erzeugt aus einem bestimmten AST-Knoten mit Hilfe der Komponenten „Metadata Management“ und „Operatortree“ einen Knoten der Operator-Baumstruktur (Strukturknoten oder DA/FA-Operator). Im Folgenden wird die Funktionalität der drei Komponenten sowie der Klasse `SemanticProcessingVisitor` zusammengefasst:

Komponente „Parser“

Abbildung 7.4 beschreibt die Funktionalität der Komponente „Parser“ mit einem UML-Aktivitätsdiagramm [HKKR05]. Die Klasse `Parser` implementiert die Syntaxanalyse (vgl. Abschnitt 6.2.3), d.h. eine von einem Benutzer formulierte SQL-MDi-Abfrage wird auf die Einhaltung der SQL-MDi-Syntaxregeln überprüft und ein Syntaxbaum (AST) erzeugt. Dafür fordert die Klasse `Parser` das jeweils nächste Symbol (Token) gemäß der SQL-MDi-Grammatik vom Token-Manager (= Scanner, implementiert lexikalische Analyse (vgl. Abschnitt 6.2.2)) an. Wird das Symbol nicht gefunden (d.h. ein Syntaxfehler wurde erkannt), wird eine `ParseException` generiert, welche den Symbolwert und die Symbolposition des fehlerhaften Symbols enthält.

Gemäß dem Aktivitätsdiagramm der Abbildung 7.4 würde das Parsing beim Auftreten einer `ParseException` beendet, was der nicht-funktionalen Anforderung „Robustheit“ (vgl. nicht-funktionale Anforderung 7 im Abschnitt 7.1.2) widerspricht. Daher unterstützt der implementierte Parser den automatischen Wiederaufsatz zur Fortsetzung des Parsing (vgl. Abschnitt 6.2.3.3). Darüber hinaus wird bei der Verarbeitung einer Produktionsregel ein AST-Knoten erzeugt, welcher die für die DA/FA-Operatoren erforderlichen Daten durch Zuweisung der entsprechenden Symbole kapselt, und in den AST eingefügt.

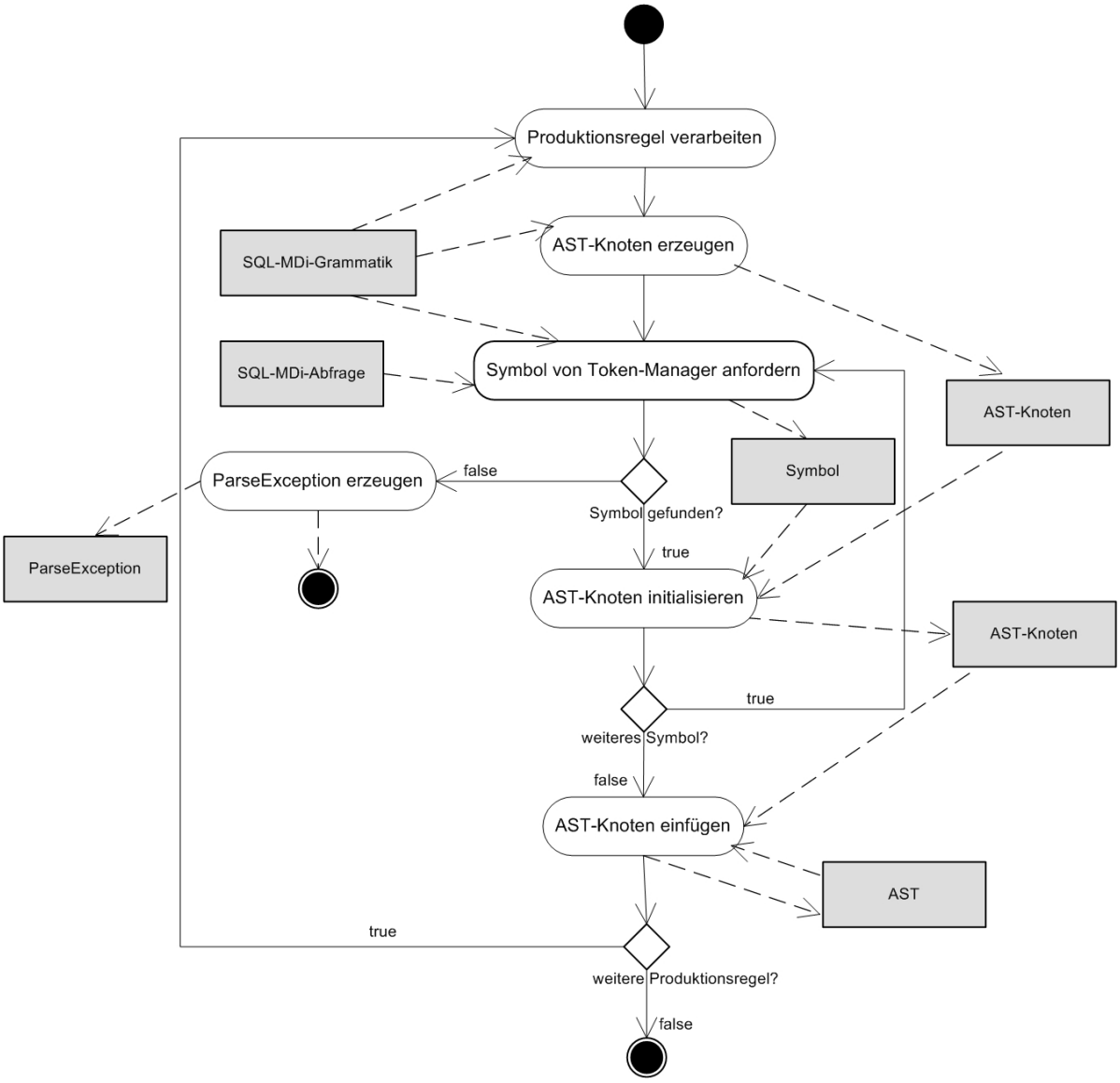


Abbildung 7.4: Funktionalität der Komponente „Parser“

Komponente „Metadata Management“

„Metadata Management“ ist die Komponente zur Semantikverarbeitung, mit welcher u.a. die Korrektheit der in einer SQL-MDi-Abfrage enthaltenen Metadaten (z.B. Informationen über Würfel, Dimensionen, Kenngrößen, etc.) durch Abgleich mit dem Metadaten-Repository überprüft wird. Die in Abbildung 7.3 dargestellte Klasse `MetaDataManager` ist eine abstrakte Basisklasse für den Repository-Zugriff, von welcher für jeden Metadatentyp separate Manager-Klassen (z.B. `CubeManager`) abgeleitet wurden. Wird ein fehlerhaftes Metadatum erkannt, so wird eine `MetaDataException` bzw. eine davon abgeleitete Metadatentyp-spezifische Exception

generiert, welche eine Angabe über das fehlerhafte Metadatum und dessen Position enthält. Wurde in der SQL-MDi-Abfrage z.B. eine fehlerhafte Würfel-Deklaration aufgrund eines falschen Namens identifiziert, so wird eine Exception vom Typ `CubeException` erzeugt.

Komponente „Operatortree“

„Operatortree“ ist die Komponente zur Semantikverarbeitung, mit welcher die Operator-Baumstruktur aufgebaut wird. Analog zu Abbildung 5.2 wird jeder Strukturknoten der Baumstruktur bzw. jeder DA/FA-Operator durch eine separate Klasse repräsentiert. Bei jedem einzufügenden Operator wird die Baumstruktur auf mögliche logische Konflikte zwischen Operatoren (vgl. Abschnitt 5.3.4.3) geprüft.

Wird ein Konflikt erkannt, wird eine Fehlermeldung mit der Klasse `QueryContextException` bzw. einer davon abgeleiteten Konflikt-spezifischen Exception generiert. Soll aufgrund einer SQL-MDi-Abfrage z.B. eine nicht-importierte Kenngröße konvertiert werden, so wird eine Exception vom Typ `MeasureConversionContextException` erzeugt (vgl. Abschnitt 5.3.4.3).

Klasse `SemanticProcessingVisitor`

Die Aufgabe der Klasse `SemanticProcessingVisitor` ist die Verarbeitung des von der Komponente „Parser“ generierten AST. Die Klasse implementiert das Visitor-Design Pattern [GHJV95] (vgl. Abschnitt 7.5.4.1) und enthält eine Menge von `visit()` Methoden, welche jeweils einen bestimmten AST-Knotentyp als Parameter akzeptiert. Somit können für jeden AST-Knotentyp bestimmte Operationen zum Erzeugen der DA/FA-Operatoren ausgeführt werden. Abbildung 7.5 visualisiert den Ablauf der Verarbeitung des AST mit Hilfe der Klasse `SemanticProcessingVisitor` als UML-Aktivitätdiagramm [HKKR05].

Im Zuge der Traversierung des AST wird für jeden AST-Knotentyp die zugeordnete `visit()` Methode aufgerufen. In dieser Methode werden zunächst die Metadaten des AST-Knotens abgefragt, d.h. die im Zuge des Parsing zugeordneten Symbole. Die Gültigkeit der Metadaten wird durch Abgleich mit den Schemainformationen des Metadaten-Repository geprüft. Wird ein Metadatum als ungültig identifiziert, so wird eine Exception vom Typ `MetaDataException` bzw. eine davon abgeleitete Metadatentyp-spezifische Exception erzeugt.

Wie aus dem Aktivitätsdiagramm der Abbildung 7.5 ersichtlich, bewirkt das Auftreten einer `MetaDataException` keinen Abbruch der Verarbeitung. Der Grund dafür liegt darin, dass der Benutzer über möglichst viele Fehler informiert werden soll, weshalb die Baumstruktur trotz fehlerhafter Metadaten aufgebaut wird (vgl. nicht-funktionale Anforderung 7 im Abschnitt 7.1.2). Die Prüfung der Metadaten sowie eine etwaige Erzeugung einer `MetaDataException` wird von der Komponente „Metadata Management“ wahrgenommen.

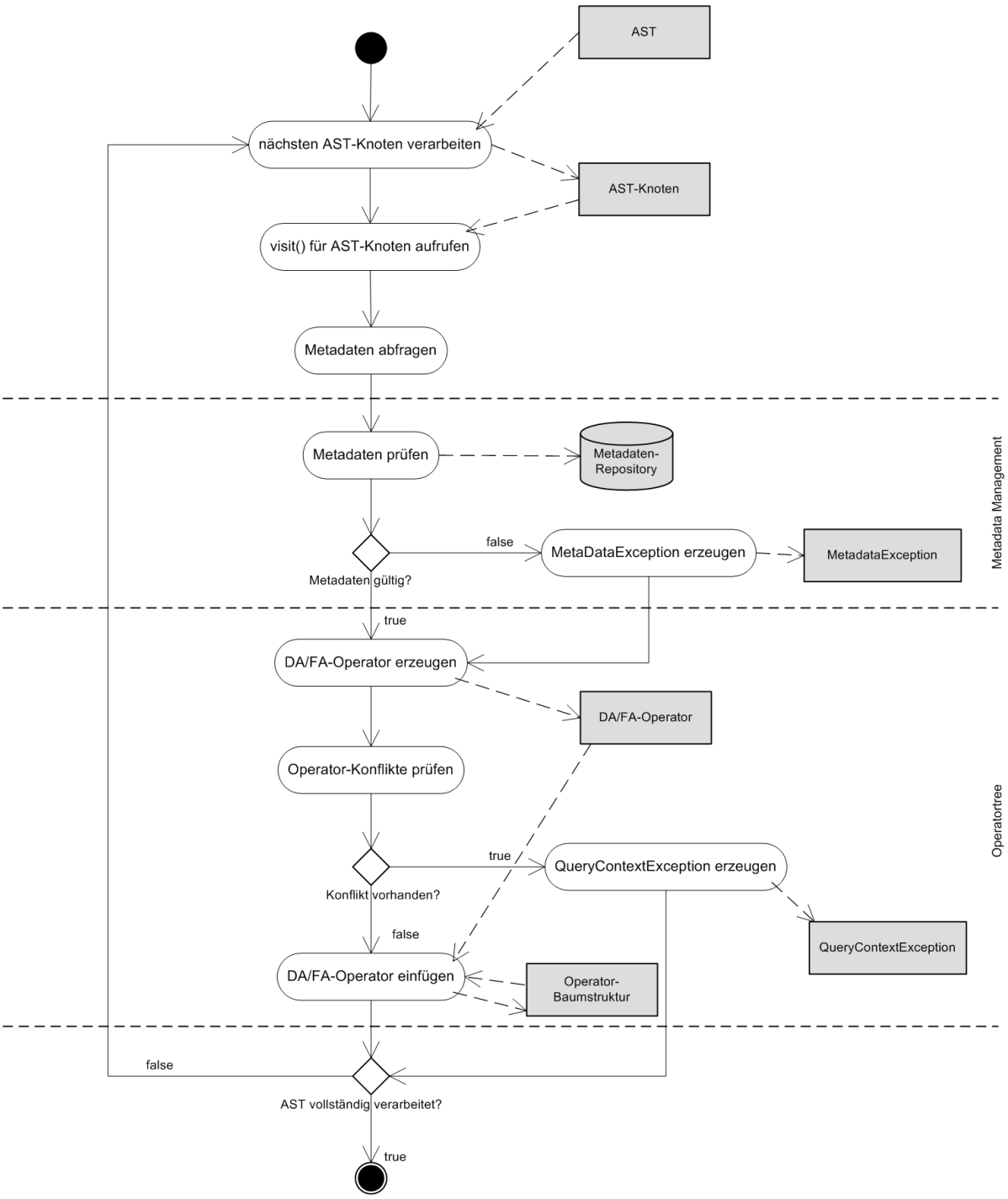


Abbildung 7.5: Verarbeitung des AST mit der Klasse `SemanticProcessingVisitor`

Im Anschluss wird ein dem jeweiligen AST-Knotentyp entsprechendes DA/FA-Operator-Objekt erzeugt, wobei in diesem Zusammenhang geprüft wird ob dieser Operator einen oder mehrere logische Konflikte (vgl. Abschnitt 5.3.4.3) in der Baumstruktur hervorrufen würde.

Wird ein Konflikt erkannt, so wird eine Exception vom Typ `QueryContextException` bzw. einer davon abgeleiteten Konflikt-spezifischen Exception erzeugt und die Verarbeitung des AST-Knotens abgebrochen; wird kein Konflikt erkannt, so wird das DA/FA-Operator-Objekt in die Baumstruktur eingefügt. Das Erzeugen, Prüfen und Einfügen von Operator-Objekten wird von der Komponente „Operatortree“ wahrgenommen.

Im folgenden Abschnitt wird die Implementierung der drei in diesem Abschnitt kurz vorgestellten Komponenten sowie der Klasse `SemanticProcessingVisitor` detaillierter erläutert.

7.5 Systemimplementierung

In diesem Abschnitt wird die Implementierung des SQL-MDi Query Parser erläutert, um Dritten die Möglichkeit zu geben sich rasch in das System einzuarbeiten und ggf. Änderungen am System durchführen zu können. Die Erläuterung der Systemimplementierung erfolgt pro Komponente, d.h. für „Parser“, „Metadata Management“ und „Operatortree“, sowie für die Klasse `SemanticProcessingVisitor`. In diesem Abschnitt wird gezeigt, wie die Erfüllung der funktionalen und nicht-funktionalen Anforderungen des Abschnitts 7.1 sichergestellt wurde.

Nachdem in der Beschreibung der Systemarchitektur die einzelnen Systemkomponenten weitestgehend als „Black-Boxes“ betrachtet wurden, werden in diesem Abschnitt Implementierungsdetails zu den Komponenten präsentiert. Mit Hilfe dieser Details und beispielhafter Demonstrationen der Query Parser-Funktionalität wird ein tieferer Einblick in die Systeminterna gewährt.

7.5.1 Komponente „Parser“

In diesem Abschnitt wird demonstriert, wie mit Hilfe des Übersetzer-Generators JavaCC (vgl. Abschnitt 7.3.6) die Komponente „Parser“ erstellt wurde. Zunächst wird gezeigt, wie der Token-Manager bzw. Scanner deklarativ erstellt wurde und wie die SQL-MDi-Grammatik in Form von Produktionsregeln einer JavaCC-Grammatik-Spezifikation bzw. der Parser implementiert wurde. Im Anschluss folgt eine Erläuterung der Anwendung des JJTree-Präprozessors zur Generierung eines AST. Am Ende dieses Abschnitts werden die Anpassung von Fehlermeldungen und der automatische Wiederaufsatz behandelt.

7.5.1.1 Klasse `TokenManager`

Die Klasse `TokenManager`, repräsentiert die lexikalische Analyse bzw. den Scanner (vgl. Abschnitt 6.2.2), in welcher eine Zerlegung einer SQL-MDi-Abfrage in eine Folge von Symbolen erfolgt. Symbole werden durch die Klasse `Token` repräsentiert, welche auch die Symbolposition für Fehlermeldungen erfasst (vgl. funktionale Anforderung 1.a im Abschnitt 7.1.1). Der

Token-Manager wird in der Grammatik-Spezifikation durch Verwendung des Produktionsregeltyps regular_expr_production für Schlüsselwörter, Namen, Sonderzeichen, etc. (vgl. Abschnitt 7.3.6) deklariert und durch Übersetzung mit dem JavaCC-Compiler automatisch erzeugt. Symbole können mit vier Varianten von regular_expr_production mit regulären Ausdrücken (vgl. Abschnitt 6.1.2.2) definiert werden: [Mic]

TOKEN:
Reguläre Ausdrücke dieses Typs beschreiben Symbole der Grammatik. Der Token-Manager erzeugt für jede Übereinstimmung mit einem regulären Ausdruck ein `Token`-Objekt, welches an den Parser übergeben wird. Dieser Typ wurde für die Definition von Schlüsselwörtern, Namen, etc. verwendet:

```
1  TOKEN: /* Schlüsselwörter */
2  {
3    < CUBE: "CUBE" >
4  | < MEASURE: "MEASURE" >
5  | < DIM: "DIM" >
6  | < PIVOT_MEASURE: "PIVOT MEASURE" >
7  | < BASED_ON: "BASED ON" >
8  ...
9  }
10 TOKEN: /* Bezeichner */
11 {
12   < IDENTIFIER: <LETTER> (<LETTER>|<DIGIT>|<SPECIAL_SIGN>)* >
13 | < #LETTER: ["_","a"-"z","A"-"Z"] >
14 | < #DIGIT: ["0"-"9"] >
15 | < #SPECIAL_SIGN: ["_","/","$","#"] >
16 /* Token mit Präfix # werden nicht an Parser übergeben */
17 ...
18 }
```

Listing 7.1: Parser: Definition von Symbolen mit TOKEN

Die Definition von Symbolen erfolgt durch Angabe eines Namens für das Symbol sowie der zugeordneten Zeichenkette (z.B. <`CUBE: "CUBE"`>). Das Symbol `IDENTIFIER` wird durch einen regulären Ausdruck beschrieben, gemäß dem dieses mit einem Buchstaben beginnen muss, gefolgt von einer beliebigen Anzahl von Buchstaben, Ziffern oder Sonderzeichen ((...)*).

SPECIAL_TOKEN:
Reguläre Ausdrücke dieses Typs beschreiben Symbole, welche keine Bedeutung für das Parsing haben, d.h. sie werden vom Parser ignoriert. Dieser Typ wird für die Definition von Kommentaren verwendet (vgl. funktionale Anforderung 1.c im Abschnitt 7.1.1):

```
1  SPECIAL_TOKEN:
2  {
3    <LINE_COMMENT: "--"(~["\r","\n"])*>
4  | <MULTI_LINE_COMMENT: "/*" (~["*"])* "*" ("*" | (~["*","/"] (~["*"])*"*"))* "/">
5  }
```

Listing 7.2: Parser: Definition von Kommentaren mit SPECIAL_TOKEN

Im Listing 7.2 werden zwei Arten von Kommentaren definiert: ein einzeiliger Kommentar, welcher mit −− eingeleitet wird und ein mehrzeiliger Kommentar, welcher zwischen /* und */ steht.

SKIP:
Übereinstimmungen mit regulären Ausdrücken dieses Typs werden vom Token-Manager ignoriert, wodurch z.B. Leerzeichen, Zeilenumbrüche, etc. oder diverse Sonderzeichen überlesen werden können (vgl. funktionale Anforderung 1.b im Abschnitt 7.1.1):

```
1  SKIP :
2  {
3    " " | "\t" | "\n" | "\r" | "^" | "!" |"§" | "$" | "%" | "&" | "?" |
4    "*" | "#" | "~" | "+" | "|" | "@" | ";" | "`" | "´" | "°" | "ä" | "ö" | "ü"
5  }
```

Listing 7.3: Parser: Definition von zu ignorierenden Zeichen mit SKIP

MORE:
Übereinstimmungen mit regulären Ausdrücken dieses Typs werden bis zur nächsten Übereinstimmung mit einem `TOKEN/SPECIAL_TOKEN` in einem Buffer zwischengespeichert. Dann werden alle Übereinstimmungen im Buffer sowie die `TOKEN/SPECIAL_TOKEN` Übereinstimmung verkettet und das neue, zusammengesetzte Symbol an den Parser übergeben. Mit dieser Variante können somit Symbole verkettet und das neue, zusammengesetzte Symbol durch den Parser verwendet werden. Dieser Typ wurde für die Erstellung des Token-Managers nicht verwendet.

Beispiel 7.1: Für die Demonstration der Funktionalität des Token-Managers wird folgende SQL-MDi-Anweisung als Input verwendet:

```
PIVOT MEASURE c2.umsatz BASED ON c2.umsatzkategorie
```

Der Token-Manager liest die Anweisung Zeichen für Zeichen ein. Sobald eine Zeichenfolge mit einer Symboldefinition des Listing 7.1 übereinstimmt, gilt das entsprechende Symbol als erkannt und kann vom Parser angefordert werden. Entsprechend der obigen Anweisung erkennt der Token-Manager zunächst das Symbol <`PIVOT_MEASURE`>, dann „c2“ als <`IDENTIFIER`>, dann „umsatz“ als <`IDENTIFIER`>, dann <`BASED_ON`>, etc.

7.5.1.2 Klasse `Parser`

Die Klasse `Parser` repräsentiert die Syntaxanalyse (vgl. Abschnitt 6.2.3) und implementiert die Produktionsregeln der SQL-MDi-Grammatik (vgl. Anhang A) um eine SQL-MDi-Abfrage auf syntaktische Korrektheit zu prüfen (vgl. funktionale Anforderung 2.a im Abschnitt 7.1.1). Für die Implementierung der Regeln können die Produktionsregeltypen javacode_production und bnf_production (vgl. Abschnitt 7.3.6) verwendet werden; für die Implementierung der SQL-MDi-Grammatik ist nur bnf_production relevant. Listing 7.4 zeigt die Implementierung der SQL-MDi-Grammatikregel für die `PIVOT MEASURE` Anweisung:

```
1  void pivot_split_measure_attr():
2  {}
3  {
4  //                cube-alias          measure-attr-name
5  <PIVOT_MEASURE> <IDENTIFIER> "." <IDENTIFIER>
6  //             cube-alias          context-dim-attr
7  <BASED_ON> <IDENTIFIER> "." <IDENTIFIER>
8  [LOOKAHEAD(2) rename_context_dim_instances()]
9  }
```

Listing 7.4: Parser: Produktionsregel für PIVOT MEASURE Anweisung

Die JavaCC-Produktionsregel des Listing 7.4 beschreibt die Syntax zur Definition einer `PIVOT MEASURE` Anweisung, indem die in Listing 7.1 definierten Symbole verwendet werden. Zu beachten ist, dass <IDENTIFIER> mit beliebigen alphanumerischen Zeichenketten übereinstimmt, weshalb dieses Symbol für alle Metadaten (z.B. Würfel, Dimensionen, Kenngrößen, etc.) verwendet wird. Welchen Metadatentyp <IDENTIFIER> konkret repräsentiert, geht aus der Reihenfolge der Anordnung hervor (siehe Kommentare im Listing 7.4).

`[LOOKAHEAD(2) rename_context_dim_instances()]` stellt einen optionalen Aufruf ([...]) einer Produktionsregel zur Definition einer `RENAME CONTEXT DIM` Anweisung dar. Das JavaCC-Schlüsselwort `LOOKAHEAD(2)` dient der Angabe der Anzahl der Vorgriffssymbole (vgl. Abschnitt 6.2.3), welche für die Erkennung dieser Regel verwendet werden. In diesem Fall wird die Anzahl der Vorgriffsymbole auf 2 (Standard: 1) gesetzt, da sowohl die Regel `rename_context_dim_instances()` als auch die auf `pivot_split_measure_attr()` folgende Regel mit '`(`' beginnen und der Parser die als nächstes anzuwendende Regel mit nur einem Vorgriffssymbol nicht eindeutig identifizieren könnte.

Der JavaCC-Compiler erzeugt beim Übersetzen der Grammatik-Spezifikation aus jeder Produktionsregel eine gleichlautende Methode in der Klasse `Parser`.

Beispiel 7.2: Für die Demonstration der Funktionalität des Parser wird erneut die SQL-MDi-Anweisung des Beispiels 7.1 als Input verwendet:

```
PIVOT MEASURE c2.umsatz BASED ON c2.umsatzkategorie
```

Der Parser fordert jeweils das nächste erkannte Symbol vom Token-Manager und prüft es auf Übereinstimmung mit der Grammatikregel des Listing 7.4. Am Beginn der Verarbeitung dieser Grammatikregel wird geprüft, ob das vom Token-Manager übergebene Symbol dem Symbol <PIVOT_MEASURE> entspricht, im Anschluss, ob das nächste übergebene Symbol ein <IDENTIFIER> ist, etc. Im Falle der obigen Anweisung, entspricht die Symbolfolge der Grammatikregel, d.h. es bestehen keine Syntaxfehler.

7.5.1.3 Abstract Syntax Tree (AST)

Der JJTree-Präprozessor erlaubt die automatische Erweiterung der Produktionsregel-Methoden der generierten Klasse `Parser` um Anweisungen zur Erzeugung eines Syntaxbaums („Abstract Syntax Tree“ (AST)) (vgl. Abschnitt 6.2.3 und funktionale Anforderung 2.b im Abschnitt

7.1.1). Dafür müssen für jene Produktionsregeln, welche eine Repräsentation im Syntaxbaum haben sollen, Knoten deklariert werden. Listing 7.5 deklariert für die Produktionsregel aus Listing 7.4 einen AST-Knoten:

```
1  void pivot_split_measure_attr() #PivotSplitMeasure:
2  {
3    Token t;
4  }
5  {
6    <PIVOT_MEASURE>
7    t=<IDENTIFIER> {jjtThis.setMCubeAlias(t);} "."
8    t=<IDENTIFIER> {jjtThis.setMeasureAttr(t);}
9    <BASED_ON>
10   t=<IDENTIFIER> {jjtThis.setDCubeAlias(t);} "."
11   t=<IDENTIFIER> {jjtThis.setDimAttr(t);}
12   [LOOKAHEAD(2) rename_context_dim_instances()]
13 }
```

Listing 7.5: Parser: Definition eines AST-Knotens

Die Anweisung `#PivotSplitMeasure` deklariert einen AST-Knoten vom Typ `ASTPivotSplitMeasure`, welcher vom Typ `SimpleNode` abgeleitet sein muss. Es besteht die Möglichkeit die Klasse `ASTPivotSplitMeasure` automatisch erstellen zu lassen und diese danach entsprechend anzupassen oder die Klasse zuvor manuell zu erstellen. Soll das Visitor-Design Pattern [GHJV95] (vgl. Abschnitt 7.5.4.1) verwendet werden, so enthält jede AST-Klasse eine Methode `accept()`. Bei der Implementierung des Query Parser wurde die zweite Variante bevorzugt, d.h. für Listing 7.5 wurde zunächst die Klasse `ASTPivotSplitMeasure` mit den Instanzvariablen `mCubeAlias`, `measureAttr`, `dCubeAlias` und `dimAttr` vom Typ `Token`, sowie den dazugehörigen `get/set`-Methoden erstellt.

Die Produktionsregel aus Listing 7.4 wurde zusätzlich um eine Variable `t` vom Typ `Token`, welche Symbole repräsentiert, und um Java-Code-Teile ({...}) erweitert. Der Variable `t` wird das zuletzt gelesene Symbol vom Typ <IDENTIFIER> zugewiesen, welches der jeweiligen Instanzvariable des Knoten-Objekts, `jjtThis`, zugewiesen wird; z.B. `t=<IDENTIFIER> jjtThis.setMCubeAlias(t);` weist `t` dem Knoten-Objekt als Würfel-Alias zu.

Hat man für alle relevanten Produktionsregeln entsprechende Knoten deklariert und übersetzt man die JavaCC-Grammatik-Spezifikation, so wird neben den im Abschnitt 7.3.6 erwähnten Dateien zusätzlich eine Datei „QueryParserVisitor.java“ erstellt. Diese Datei enthält die Deklaration eines Interface für das Visitor-Design Pattern [GHJV95], welches für jeden Knoten-Typ eine Schnittstelle als `visit()` Methode anbietet (vgl. Abschnitt 7.5.4.1). Für die Produktionsregel aus Listing 7.5 würde die Methoden-Schnittstelle `public Object visit(ASTPivotSplitMeasure node, Object data);` erzeugt werden.

Der JavaCC-Compiler ergänzt beim Übersetzen der Grammatik-Spezifikation die Produktionsregel-Methoden der Klasse `Parser` automatisch um Anweisungen zum Aufbau eines AST. Am Beginn einer Methode wird ein Knoten-Objekt erzeugt, das später mit jenen Symbolen initialisiert wird, welche die für die Erzeugung eines DA/FA-Operators erforderlichen Daten repräsentieren. Am Ende einer Methode wird das Knoten-Objekt in den AST eingefügt.

Beispiel 7.3: Für die Demonstration der Erzeugung eines AST-Knotens wird erneut die SQL-MDi-Anweisung des Beispiels 7.1 herangezogen:

```
PIVOT MEASURE c2.umsatz BASED ON c2.umsatzkategorie
```

Wird für obige SQL-MDi-Anweisung die Grammatikregel des Listing 7.5 verwendet, so wird das AST-Knoten-Objekt `jjtThis` vom Typ `ASTPivotSplitMeasure` erzeugt:

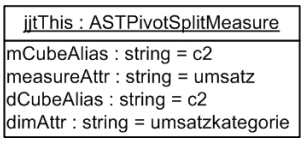


Abbildung 7.6: Parser: `ASTPivotSplitMeasure`-Knoten

Der Vaterknoten für den `ASTPivotSplitMeasure`-Knoten ergibt sich aus der Verarbeitungsreihenfolge für die Grammatikregeln. Dadurch, dass der Parser die Grammatikregeln top-down verarbeitet, wird der `ASTPivotSplitMeasure`-Knoten als Kindknoten eines `ASTCube`-Knotens in den AST eingefügt. Die dem `ASTCube`-Knoten zugeordnete Grammatikregel ist der des Listing 7.5 als nächste übergeordnet.

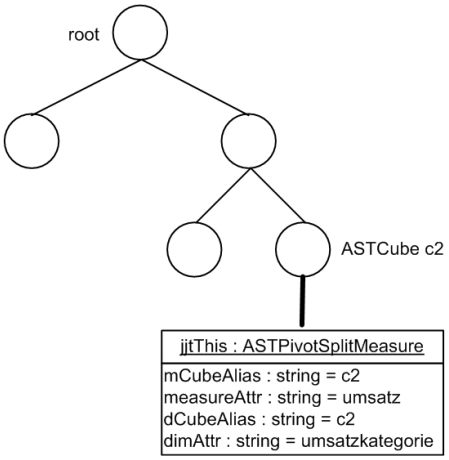


Abbildung 7.7: Parser: Einfügen in den AST

7.5.1.4 Anpassung von Fehlermeldungen und automatischer Wiederaufsatz

Wird während des Parsing ein Syntaxfehler erkannt, so wird eine Fehlermeldung vom Typ `ParseException` erzeugt. Eine solche Fehlermeldung enthält bereits standardmäßig den Symbolwert und die Symbolposition des fehlerhaften Symbols sowie das(die) erwartete(n) Symbol(e) (vgl. funktionale Anforderung 2.c im Abschnitt 7.1.1). Die Fehlerausgabe des SQL-MDi Query Parser sollte zu den erwarteten Symbolen die zugeordneten SQL-MDi-Anweisungen

umfassen. Um dies zu erreichen, wurde die Methode `getMessage()` der Klasse `ParseException` entsprechend modifiziert:

```
1  public String getMessage(){
2    ...
3    StringBuffer expected = new StringBuffer();
4    ...
5    for (int j = 0; j < expectedTokenSequences[i].length; j++) {
6    ...
7    switch(expectedTokenSequences[i][j]){
8      case 25: expected.append("(Keyword of DEFINE (GLOBAL) CUBE clause)");
9               break;
10     case 26: expected.append("(Keyword of DEFINE GLOBAL CUBE clause)");
11              break;
12     case 27: expected.append("(Keyword of DEFINE CUBE clause)");
13              break;
14     ...
15   }
16   ...
17 }
```

Listing 7.6: Parser: Anpassung von Fehlermeldungen

Die Variable `expected` wird verwendet um eine Fehlermeldung zu erzeugen. Jeder Eintrag des Arrays `expectedTokenSequences` ist ein Integer-Array, welches die Symbolcodes der erwarteten Symbole enthält. Die Symbolcodes ergeben sich aus der Reihenfolge, in der die Symbole in der Grammatik-Spezifikation deklariert wurden (vgl. Abschnitt 7.5.1.1). Im Listing 7.6 wird über die Integer-Arrays iteriert und für den jeweiligen Symbolcode die entsprechende SQL-MDi-Anweisung zur Fehlermeldung hinzugefügt. Beispiel 7.4 zeigt die Ausgabe einer Fehlermeldung durch die modifizierte `getMessage()` Methode.

Beispiel 7.4: Beginnt die in den Beispielen 7.1 - 7.3 verwendete SQL-MDi-Anweisung mit `PIVOT` statt mit `PIVOT MEASURE`, so wird folgende Fehlermeldung erzeugt:

Encountered "PIVOT" at line 5, column 1.
Was expecting one of:
"GLOBAL CUBE" (Keyword of DEFINE GLOBAL CUBE clause)
"CUBE" (Keyword of DEFINE CUBE clause)
"PIVOT MEASURE" (Keyword of PIVOT (Split measure attribute) clause)
"PIVOT MEASURES" (Keyword of PIVOT (Merge measure attributes) clause)
"DIM" (Keyword of DIM IMPORT clause or DIM MAPPING clause)

Your query contains some syntax-errors. Please correct those errors and restart parsing!

Die von JavaCC standardmäßig implementierte Fehlerbehandlung bewirkt einen Abbruch des Parsing sobald ein Fehler erkannt wird. Wie im Abschnitt 6.2.3 erläutert, ist diese rudimentäre Fehlerbehandlung nicht wünschenswert, weshalb der Query Parser für das Verfahren „Wiederaufsatz mit speziellen Fangsymbolen“ erweitert wurde (vgl. nicht-funktionale Anforderung 7 im Abschnitt 7.1.2). Bei diesem Verfahren wird eine Symbolfolge im Fehlerfall solange überlesen, bis ein definiertes Fangsymbol (Schlüsselwort) auftritt, d.h. ab dann ist der Parser wieder in einem gültigen Zustand und das Parsing kann fortgesetzt werden. Listing 7.7 zeigt die Berücksichtigung des Wiederaufsatzes für eine Produktionsregel der JavaCC-Grammatik-Spezifikation:

```
1   void cube_specs():
2   {}
3   {
4     try{
5       cube_spec()
6     }catch(ParseException e){
7       new ErrorHandler(e,CUBE,GLOBAL_CUBE,MERGE_DIMENSIONS,MERGE_CUBES,EOF).skipTo_1();
8     }
9     ...
10  }
```

Listing 7.7: Parser: Automatischer Wiederaufsatz

Listing 7.7 zeigt einen Ausschnitt der Produktionsregel für die `DEFINE CUBE` Anweisung. Der Aufruf der Regel `cube_spec()` wurde in einen `try-catch` Block geschachtelt, wodurch eine aufgrund eines Fehlers in dieser Regel generierte `ParseException` gefangen wird und die Fehlerbehandlung mit der Methode `skipTo_1()` eines `ErrorHandler`-Objekts angestoßen wird. Der Konstruktor des `ErrorHandler`-Objekts erhält als Parameter das `ParseException`-Objekt `e` und eine Menge von (Fang)Symbolen vom Typ `Token`. In der `skipTo_1()` Methode wird die Symbolfolge mit der Methode `getNextToken()` solange überlesen bis entweder ein `CUBE`, `GLOBAL_CUBE`, `MERGE_DIMENISONS`, `MERGE_CUBES` oder ein `EOF` (end-of-file) Symbol erkannt wird. Ab diesem Zeitpunkt wird die Analyse fortgesetzt bzw. beendet (wenn ein `EOF` Symbol erkannt wurde).

Durch Übersetzung der Grammatik-Spezifikation mit dem JavaCC-Compiler werden die Produktionsregel-Methoden der Klasse `Parser` automatisch um die entsprechenden Anweisungen zur Fehlerbehandlung ergänzt.

7.5.2 Komponente „Metadata Management“

Dieser Abschnitt behandelt das Management der in einer SQL-MDi-Abfrage enthaltenen Metadaten, z.B. Informationen über Würfel, Dimensionen, Kenngrößen, etc. Es muss insbesondere die Korrektheit der Metadaten einer Abfrage überprüft werden, d.h. es muss sichergestellt werden, dass beispielsweise ein Würfel „Verkauf“ tatsächlich im definierten Komponenten-DWS existiert (vgl. funktionale Anforderung 3.a.i im Abschnitt 7.1.1). Bevor die Überprüfung der Metadaten und die weiteren Aufgaben der Komponente „Metadata Management“ erläutert werden, ist es erforderlich das Design der Metadaten zu veranschaulichen.

7.5.2.1 Metadaten-Design

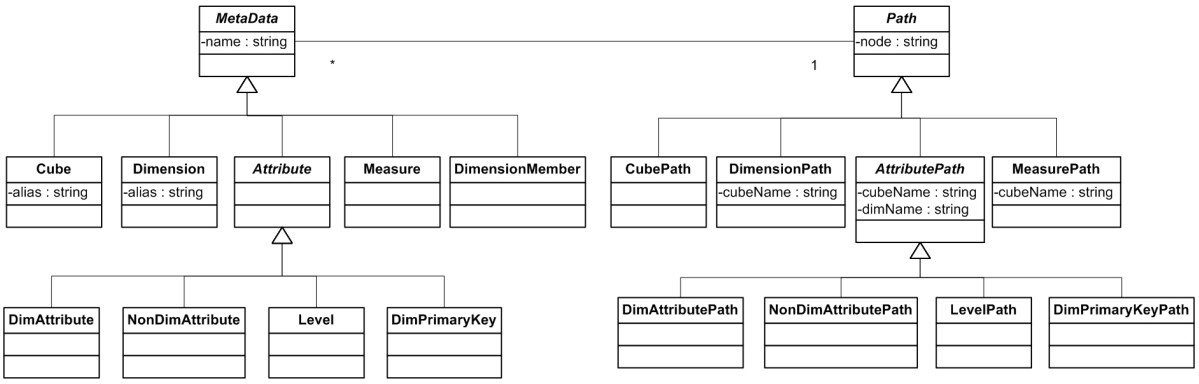


Abbildung 7.8: Metadata Management: Metadaten-Design

Das in Abbildung 7.8 dargestellte UML-Klassendiagramm [HKKR05] enthält zwei Vererbungshierarchien. Für jeden in einer SQL-MDi-Abfrage möglicherweise vorkommenden Metadatentyp existiert eine separate Klasse, welche von der abstrakten Klasse `MetaData` abgeleitet ist. Bei der Klasse `Attribute` handelt es sich wiederum um eine abstrakte Klasse, von welcher Klassen für Metadaten-Objekte einer Dimension abgeleitet sind:

- `Cube` repräsentiert einen lokalen Würfel eines Komponenten-DWS bzw. einen globalen Würfel des FDWS.
- `Dimension` repräsentiert eine lokale oder globale Dimension, welche einem bestimmten Würfel zugeordnet ist.
- `DimAttribute` repräsentiert ein dimensionales Attribut einer Dimension bzw. ein dimensionales Attribut einer Fakttabelle.
- `NonDimAttribute` repräsentiert ein nicht-dimensionales Attribut einer Dimension.
- `Level` repräsentiert eine Klassifikationsstufe einer Dimension, wobei zwischen `Level` und `DimAttribute` eine 1:1-Beziehung besteht, d.h. SQL-MDi betrachtet dimensionale Attribute und Klassifikationsstufen als äquivalent (vgl. Abschnitt 2.3)
- `DimPrimaryKey` repräsentiert den Primärschlüssel einer Dimension bzw. einer Dimensionstabelle.
- `Measure` repräsentiert ein Kenngrößen-Attribut eines Würfels.
- `DimensionsMember` wird verwendet um Instanzen eines dimensionalen Kontextattributs darzustellen (für `PIVOT` Anweisungen, vgl. Abschnitt 5.2.1).

Unabhängig vom konkreten Typ enthält jedes `MetaData`-Objekt einen Namen sowie eine Referenz auf ein äquivalentes `Path`-Objekt (z.B. hat ein `Cube`-Objekt eine Referenz auf ein `CubePath`-Objekt), welches den Pfad für die eindeutige Lokalisierung in einem lokalen Komponenten-DWS darstellt. Unabhängig vom konkreten Typ enthält jedes `Path`-Objekt die Bezeichnung des DW-Knotens (z.B. einen Alias für den Servernamen).

Für `MetaData`-Objekte, welche durch die zusätzliche Angabe des Würfels eindeutig identifiziert werden können, nämlich Dimensionen, Kenngrößen und dimensionale Attribute einer Fakttabelle, enthält das `Path`-Objekt den Namen des entsprechenden Würfels (`cubeName`); für `MetaData`-Objekte, welche darüber hinaus die Angabe einer Dimension erfordern um eindeutig identifiziert werden zu können, nämlich (nicht-)dimensionale Attribute, Klassifikationsstufen, und der Primärschlüssel einer Dimension, enthält das `Path`-Objekt zusätzlich den Namen der entsprechenden Dimension (`dimName`). Für `DimensionMember`-Objekte ist die Zuordnung eines `Path`-Objekts nicht erforderlich. Für ein `Cube`- bzw. `Dimension`-Objekt kann zusätzlich ein Alias (`alias`) definiert werden, welcher in SQL-MDi für die Zuordnung eines `MetaData`-Objekts zu einem bestimmten Würfel bzw. zu einer Dimension verwendet wird.

Beispiel 7.5: Das folgende Instanzdiagramm zeigt ein `Measure`-Objekt für die Kenngröße „Umsatz“ im Würfel „Verkauf“ des Knotens „mfuB“:

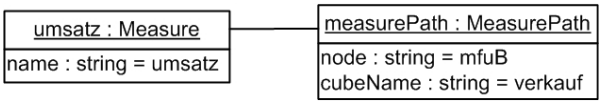


Abbildung 7.9: Metadata Management: `Measure`-Objekt mit Pfad-Angabe

Basierend auf dem eben vorgestellten Metadaten-Design werden im folgenden Abschnitt die Aufgaben der Komponente „Metadata Management“ sowie deren Realisierung erläutert.

7.5.2.2 Metadaten-Management

Den Kern der Komponente „Metadata Management“ bilden die abstrakte Klasse `MetaDataManager` und die von dieser abgeleiteten Metadatentyp-spezifischen Manager-Klassen; für jeden Metadatentyp aus Abbildung 7.8 existiert eine eigene Manager-Klasse (z.B. `CubeManager`, `DimensionManager`, etc.). Abbildung 7.10 visualisiert einen Ausschnitt aus der Vererbungshierarchie dieser Manager-Klassen und führt die von dieser Komponente wahrgenommenen Aufgaben an:

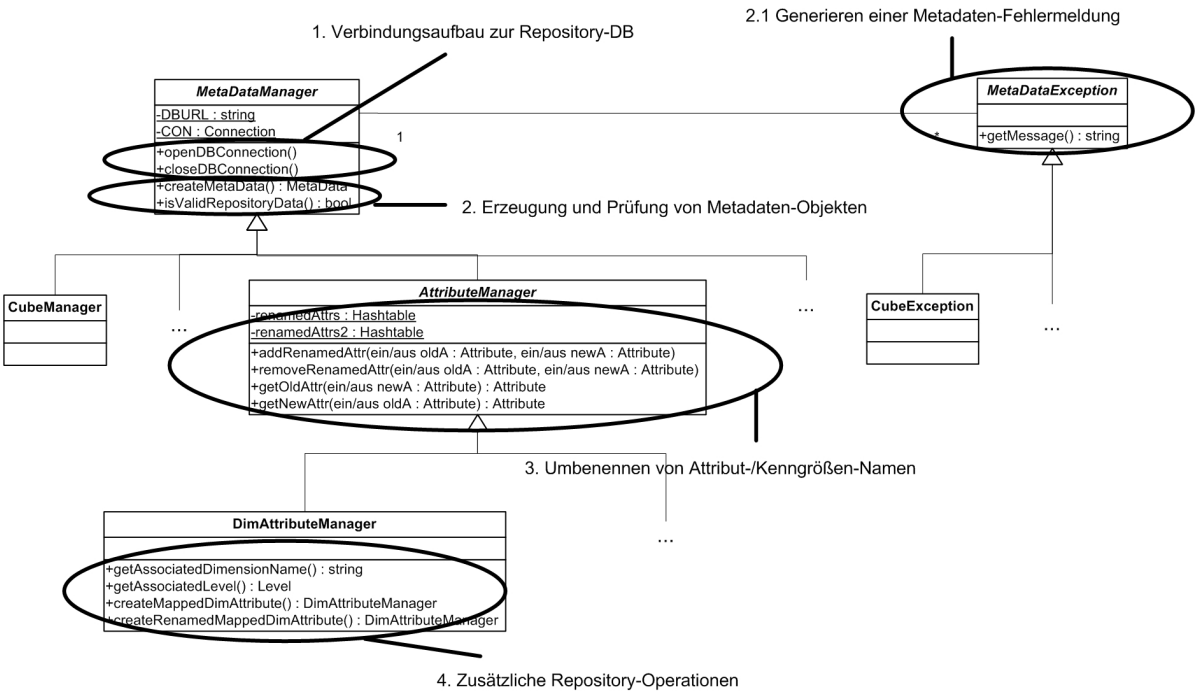


Abbildung 7.10: Metadata Management: Metadaten-Management-Design

Die Vererbungshierarchie der Metadaten-Manager-Klassen weist dieselbe Struktur auf wie jene der Metadaten (vgl. Abbildung 7.8), d.h. für jeden Metadatentyp existiert eine separate Manager-Klasse, welche Operationen für diesen Typ implementiert. Darüber hinaus kann jedes `MetaDataManager`-Objekt eine Menge von äquivalenten `MetaDataException`-Objekten erzeugen (z.B. erzeugt ein `CubeManager` eine Menge von `CubeExceptions`). Diese Exceptions repräsentieren Fehlermeldungen bei ungültigen Metadaten und enthalten, analog zur `ParseException` (vgl. Abschnitt 7.5.1.4), das fehlerhafte Metadatum sowie dessen Position in der SQL-MDi-Abfrage (vgl. funktionale Anforderung 3.a.ii im Abschnitt 7.1.1). Im Folgenden werden die in Abbildung 7.10 dargestellten Aufgaben der Komponente „Metadata Management“ erläutert:

1. Verbindungsaufbau zur Repository-Datenbank:
   Um die Korrektheit der Metadaten zu prüfen bzw. um weitere Metadaten-Operationen auszuführen muss eine Datenbankverbindung zum Metadaten-Repository aufgebaut werden. Methoden zum Öffnen und Schließen der Datenbankverbindung wurden in der abstrakten Klasse `MetaDataManager` implementiert.
2. Erzeugung und Prüfung von Metadaten-Objekten:
   Die Hauptaufgabe der Manager-Klassen ist die Erzeugung eines äquivalenten `MetaData`-Objekts (z.B. erzeugt ein `CubeManager` ein `Cube`-Objekt) aus einem Symbol, welches den Namen des Metadatums darstellt, und aus einer Menge von Symbolen, welche zusammen den Lokalisierungspfad (`Path`-Objekt) bilden. Diese Aufgabe wird durch die Methode `createMetaData()` realisiert.

   Darüber hinaus ist die Prüfung der Korrektheit der Metadaten eine wichtige Anforderung an den Query Parser (vgl. Abschnitt 7.1.1), welche mit der Methode `isValidRepository-`

`Data()` umgesetzt wurde. Diese Methode wurde derart implementiert, dass ein von der Methode `createMetaData()` erzeugtes `MetaData`-Objekt unter Verwendung des von ihm referenzierten `Path`-Objekts im Repository gesucht wird. Wird das `MetaData`-Objekt gefunden, gilt es als gültig; wird es hingegen nicht gefunden, so wird eine Exception vom Typ `MetaDataException` bzw. einer davon abgeleiteten Metadatentyp-spezifischen Exception generiert (Aufgabe 2.1 in Abbildung 7.10), welche durch Initialisierung mit den Symbolen der Manager-Klasse die Fehlerposition inkludiert (vgl. Beispiel 7.6).

Beispiel 7.6: Die Kenngröße „Umsatz1“ existiert im Würfel „Verkauf“ des Knotens „mfuB“ nicht:

```
Encountered 'mfuA::verkauf.umsatz1' at line 3, column 13
The specified Measure does not exist ...
```

Abschließend sei noch angemerkt, dass jede Manager-Klasse nur `MetaData`-Objekte des ihr zugeordneten Typs prüfen bzw. erzeugen kann (vgl. Design Pattern „Factory Method“ in [GHJV95]).

3. Umbenennen von Attribut-/Kenngrößen-Namen:
   Um die Benutzerfreundlichkeit des Query Parser zu erhöhen, soll der Benutzer die Möglichkeit haben sowohl die alten als auch neuen Attribut-/Kenngrößen-Namen zu verwenden, falls er diese in einer `DIM`, `MAP LEVELS` oder `MATCH ATTRIBUTES` bzw. einer `MEASURE` Anweisung umbenannt hat (vgl. Abschnitt 5.2). Um dies zu realisieren wurden in den Klassen `MeasureManager` und `AttributeManager` Hash-Tabellen implementiert, welche eine Abbildung der neuen auf die alten Namen (z.B. `renamedAttrs`) bzw. der alten auf die neuen Namen (z.B. `renamedAttrs2`) ermöglichen.

   Soll nun die Korrektheit eines Metadatums mit der Methode `isValidRepositoryData()` festgestellt werden, so wird im Falle eines negativen Ergebnisses der Repository-Prüfung zusätzlich geprüft, ob es sich möglicherweise um einen umbenannten, neuen Namen handelt.

4. Zusätzliche Repository-Operationen:
   Neben der Prüfung der Korrektheit der Metadaten und der Erzeugung von `MetaData`-Objekten sind für manche Metadatentypen zusätzliche Operationen erforderlich, welche mit Hilfe des Metadaten-Repository ausgeführt werden. Die Klasse `DimAttributeManager` unterstützt beispielsweise Operationen zur Auffindung des Namens der Dimension eines dimensionalen Attributes einer Fakttabelle (`getAssociatedDimensionName()`) oder zur Erzeugung des einem dimensionalen Attribut entsprechenden `Level`-Objekts (`getAssociatedLevel()`).

   Darüber hinaus werden für die Prüfung von globalen Metadaten und Kontextbedingungen für manche Metadatentypen separate Methoden zur Erzeugung von `MetaData`-Objekten implementiert (z.B. `createMappedDimAttribute()` oder `createRenamedMappedDimAttribute()`).

   Die Erzeugung mancher DA/FA-Operatoren erfordert zudem weitere Schemainformation aus dem Repository (z.B. `LevelManager`: Ermitteln der zu löschenden Klassifikationsstufe(n) aufgrund einer `MAP LEVELS` Anweisung). Derartige zusätzliche Repository-Operationen sind auch in den Klassen `DimensionManager`, `LevelManager` und `MeasureManager` verfügbar.

7.5.3 Komponente „Operatortree“

Die Aufgabe der Komponente „Operatortree“ ist die Bereitstellung der Infrastruktur, welche erforderlich ist um eine vollständig verarbeitbare Operator-Baumstruktur zu erzeugen (vgl. funktionale Anforderung 3.b.i im Abschnitt 7.1.1). Zunächst wird das physische Design der im Abschnitt 5.3.4.1 präsentierten logischen Baumstruktur erläutert. Anschließend wird demonstriert, wie die DA/FA-Operatoren unter Berücksichtigung der im Abschnitt 5.3.4.2 vorgestellten Regeln in die Baumstruktur eingefügt werden. Am Ende dieses Abschnitts wird erläutert, wie mögliche logische Konflikte zwischen DA/FA-Operatoren (vgl. Abschnitt 5.3.4.3) erkannt und behandelt werden.

7.5.3.1 Physisches Design der Operator-Baumstruktur

Das physische Design der Operator-Baumstruktur folgt der im Abschnitt 5.3.4.1 präsentierten logischen Struktur, d.h. für alle Strukturknoten und DA/FA-Operatoren wurden entsprechende Klassen implementiert. Die Parameter der DA/FA-Operatoren werden in den Klassen als Instanzvariablen abgebildet. Aufgrund der Äquivalenz des logischen und physischen Designs der Operator-Baumstruktur wird auf eine Erläuterung der Klassen für die Strukturknoten und DA/FA-Operatoren verzichtet. Stattdessen werden die allen Strukturknoten und DA/FA-Operatoren gemeinsamen Eigenschaften beschrieben. Abbildung 7.11 visualisiert die unterschiedlichen, abstrakten Klassen für die Baumknoten:

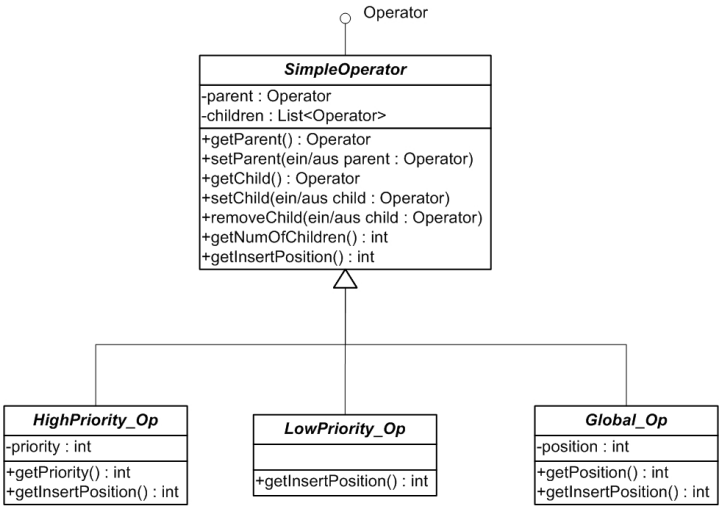


Abbildung 7.11: Operatortree: Abstrakte Basisklassen der Operator-Baumstruktur

Die Klasse `SimpleOperator` implementiert die im Interface `Operator` definierten Schnittstellen. Jeder Knoten der Operator-Baumstruktur hat eine Referenz auf einen Vaterknoten (`parent`) (außer der Wurzelknoten) und eine Liste von Kindknoten (`children`) (außer den Blättern, welche den DA/FA-Operatoren entsprechen).

Neben diversen Methoden zur Verwaltung des Vaterknotens und der Kindknoten besitzt jeder Knoten eine Methode `getInsertPosition()`. Mit Hilfe dieser Methode kann für jeden Knoten die optimale Einfügeposition in der Liste der Kindknoten des Vaterknotens ermittelt werden. Da die optimale Einfügeposition nicht für jeden Knoten gleich zu ermitteln ist, wird diese Methode in den von `SimpleOperator` abgeleiteten abstrakten Klassen, nämlich `HighPriority_Op`, `LowPriority_Op` und `Global_Op` sowie ggf. in den konkreten Operator-Klassen, überschrieben.

7.5.3.2 Einfügen von DA/FA-Operatoren

Mit Hilfe der in Abbildung 7.11 dargestellten Vererbungshierarchie wird das Einfügen der Operatoren bzw. die Arbeitsweise der Methode `getInsertPosition()`, unter Bezugnahme auf die im Abschnitt 5.3.4.2 erläuterten theoretischen Grundlagen, behandelt.

Durch die unterschiedlichen Implementierungen der Methode `getInsertPosition()` für die verschiedenen Typen von Operatoren sowie für spezifische Operatoren, kann eine korrekte und effiziente Verarbeitung der Operator-Baumstruktur durch eine Prozessor-Komponente erreicht werden. Für derartige Optimierungen wäre im Übersetzerbau eine eigene Phase „Optimierung“ vorgesehen (vgl. Abschnitt 6.2.1). Der SQL-MDi Query Parser führt diese Optimierungen jedoch im Zuge der Semantikverarbeitung aus, welche das Ziel hat eine korrekt verarbeitbare Operator-Baumstruktur zu erzeugen. Dadurch, dass für die Optimierung die bestehende Methode `getInsertPosition()` genutzt werden kann, erscheint eine separate Phase für die Optimierung als nicht zweckmäßig.

Operatoren vom Typ `HighPriority_Op`
Um eine Optimierung der Baumstruktur auf logischer Ebene zu erreichen, werden jene Klassen, welche DA/FA-Operatoren repräsentieren, die die Ergebnismenge verkleinern (σ, π, ξ, λ und α), von `HighPriority_Op` abgeleitet (vgl. funktionale Anforderung 3.b.ii im Abschnitt 7.1.1). Diese Klassen haben eine zusätzliche Instanzvariable (`priority`) für die Festlegung der Auswertungspriorität, mit welcher bestimmte DA/FA-Operatoren bevorzugt werden können. Die Methode `getInsertPosition()` wird in der Klasse `HighPriority_Op` derart überschrieben, dass Operatoren dieses Typs, unter Berücksichtigung ihrer Auswertungspriorität, am Beginn der Liste der Kindknoten des Vaterknotens eingefügt werden.

Beispiel 7.7: Um das Einfügen eines Operators vom Typ `HighPriority_Op` zu veranschaulichen, wird folgender Ausschnitt einer teilweise konstruierten Operator-Baumstruktur angenommen:

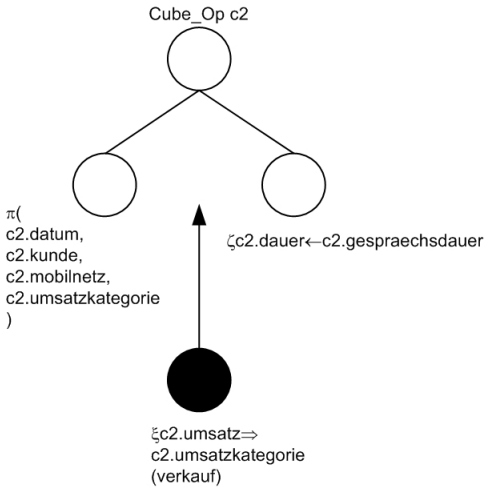


Abbildung 7.12: Operatortree: Einfügen eines Operators vom Typ `HighPriority_Op`

Abbildung 7.12 zeigt einen Ausschnitt der Operator-Baumstruktur für den Strukturknoten c2 vom Typ `Cube_Op`, welcher einen π Operator (Typ `HighPriority_Op`) und einen ζ Operator (Typ `LowPriority_Op`) als Kindknoten enthält. Der einzufügende ξ Operator ist vom Typ `HighPriority_Op` und bezüglich der Verarbeitungsreihenfolge zu bevorzugen. Die `getInsertPosition()`-Implementierung ermittelt als Einfügeposition jene vor dem ζ Operator und hinter dem π Operator. Der Grund dafür liegt darin, dass der π Operator eine höhere Priorität wie der ξ Operator besitzt. (vgl. Abschnitt 5.3.4.2)

Operatoren vom Typ `LowPriority_Op`

ζ Operatoren können Abhängigkeiten zu anderen Operatoren aufweisen, da letztere die ursprünglichen Metadaten-Namen verwenden. Aus diesem Grund müssen ζ Operatoren zuletzt angewendet werden. Jene Klassen, welche einen ζ Operator repräsentieren, werden von `LowPriority_Op` abgeleitet. Die Methode `getInsertPosition()` wird in der Klasse `LowPriority_Op` derart überschrieben, sodass ζ Operatoren am Ende der Liste der Kindknoten des Vaterknotens eingefügt werden um eine vollständige Verarbeitung der Operator-Baumstruktur sicherzustellen.

Beispiel 7.8: Um das Einfügen eines Operators vom Typ `LowPriority_Op` zu veranschaulichen, wird Beispiel 7.7 fortgeführt:

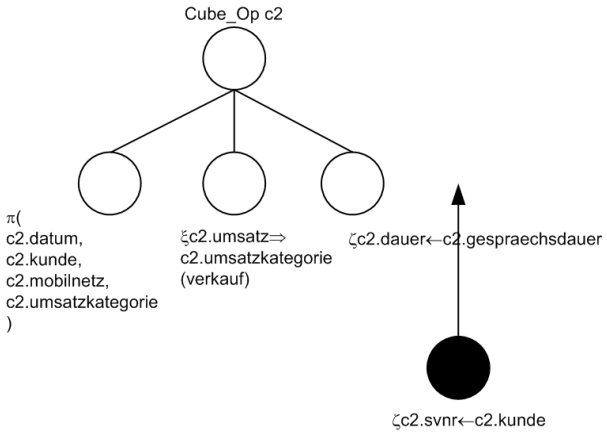


Abbildung 7.13: Operatortree: Einfügen eines Operators vom Typ `LowPriority_Op`

In der Abbildung 7.13 soll ein ζ Operator in die Operator-Baumstruktur eingefügt werden. Der ζ Operator ist vom Typ `LowPriority_Op`, dessen `getInsertPosition()`-Implementierung als Einfügeposition das Ende der Liste der Kindknoten des Knotens c2 ermittelt. (vgl. Einfüge-Regel 2.1 im Abschnitt 5.3.4.2)

Operatoren vom Typ `Global_Op`

Jene Klassen, welche global auszuwertende Operatoren (z.B. μ) repräsentieren, werden von `Global_Op` abgeleitet. Für diese Operatoren ist es möglich eine fixe Einfügeposition in die Liste der Kindknoten des Vaterknotens (des globalen Würfels) als Instanzvariable (`position`) festzulegen. Daher wird die Methode `getInsertPosition()` in der Klasse `Global_Op` derart überschrieben, dass Operatoren dieses Typs gemäß ihrer Position eingefügt werden.

Beispiel 7.9: Um das Einfügen eines Operators vom Typ `Global_Op` zu veranschaulichen, wird folgender Ausschnitt einer teilweise konstruierten Operator-Baumstruktur angenommen:

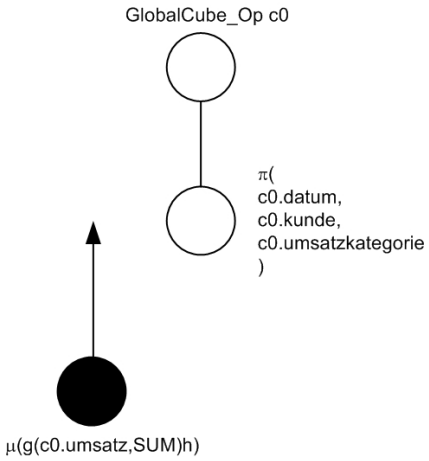


Abbildung 7.14: Operatortree: Einfügen eines Operators vom Typ `Global_Op`

Abbildung 7.14 zeigt einen Ausschnitt der Operator-Baumstruktur für den Strukturknoten c0 vom Typ `GlobalCube_Op`, welcher einen π Operator (Typ `HighPriority_Op`) als Kindknoten enthält. Die `getInsertPosition()`-Implementierung ermittelt für den μ Operator als Einfügeposition jene vor dem π Operator, da der Wert der Instanzvariable `position` von μ kleiner ist als jener von π.

Operatoren vom Typ `SimpleOperator`
All jene Klassen, welche Operatoren repräsentieren, die nicht unter eine der drei zuvor erläuterten Kategorien subsummierbar sind, werden von der Klasse `SimpleOperator` abgeleitet. Die `getInsertPosition()`-Implementierung dieser Klasse ermittelt die Einfügeposition von Operatoren dieses Typs unter Berücksichtigung bereits eingefügter Operatoren der Typen `HighPriority_Op` und `LowPriority_Op`.

Spezifische DA/FA-Operatoren
Die Methode `getInsertPosition()` wird in jenen Klassen überschrieben, deren zugeordneter DA/FA-Operator nicht unabhängig von allen anderen Operatoren angewendet werden kann. So muss z.B. ein $\gamma_{M'\ \theta}(c)$ Operator links vor einem $\xi_{M' \Rightarrow A'}(c)$ Operator als Kindknoten eines Würfel-Knotens c eingefügt werden, wenn $M'(\gamma) = M'(\xi)$. In den jeweiligen Methoden-Implementierungen wird die Liste der Kindknoten des Vaterknotens dahingehend analysiert, ob ein Operator existiert, vor oder nach dem der aktuelle Operator zwingend einzufügen ist.

Beispiel 7.10: Um das Einfügen eines Operators vom Typ `SimpleOperator` zu veranschaulichen, wird Beispiel 7.8 fortgeführt:

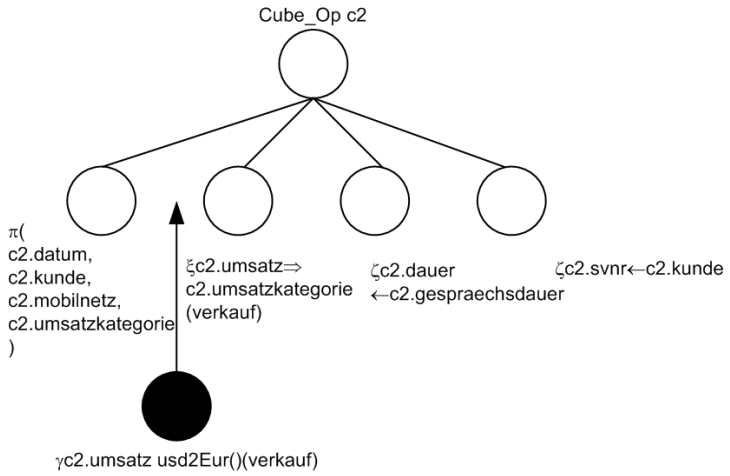


Abbildung 7.15: Operatortree: Einfügen eines Operators vom Typ `SimpleOperator`

In der Abbildung 7.15 soll ein γ Operator (vom Typ `SimpleOperator`) in die Operator-Baumstruktur eingefügt werden. Grundsätzlich müsste dieser Operator hinter den beiden Operatoren vom Typ `High-Priority_Op` (π und ξ) eingefügt werden. Da die zu konvertierende Kenngröße c2.umsatz nach Anwendung des ξ Operators jedoch nicht mehr vorhanden wäre, ermittelt die `getInsertPosition()`-Implementierung des γ Operators als Einfügeposition jene vor dem ξ Operator, damit die Konvertierung zuvor erfolgen kann. (vgl. Einfüge-Regel 7.1 im Abschnitt 5.3.4.2)

7.5.3.3 Konfliktprüfung für DA/FA-Operatoren

Wie in Abschnitt 5.3.4.3 erläutert, können bestimmte Kombinationen von DA/FA-Operatoren eine vollständige Verarbeitung der Operator-Baumstruktur verhindern (vgl. Beispiel 7.11). Daher muss der SQL-MDi Query Parser zwischen Operatoren bestehende logische Konflikte erkennen und eine aussagekräftige Fehlermeldung generieren (vgl. funktionale Anforderung 3.b.iii im Abschnitt 7.1.1). Realisiert wurde dies, indem jene Klassen, welche eventuell in einem Konflikt stehende DA/FA-Operatoren repräsentieren, eine Methode `checkQueryContext()` implementieren.

Diese Methode wird im Zuge des Einfügens eines potentiell konfliktären Operators (vgl. Abbildung 5.3) am Beginn der Methode `getInsertPosition()` aufgerufen, d.h. bevor die optimale Einfügeposition für den Operator ermittelt wird. `checkQueryContext()` analysiert die bisher konstruierte Baumstruktur auf potentiell konfliktäre Operatoren. Wurde ein konfliktärer Operator identifiziert, werden dessen relevante Parameter mit jenen des einzufügenden Operators verglichen. Wurde auf diese Weise ein Konflikt erkannt, wird eine Exception vom Typ `QueryContextException` bzw. einer davon abgeleiteten Konflikt-spezifischen Exception gene-

riert, deren Fehlermeldung Information über den einzufügenden Operator umfasst und Hinweise auf die Konfliktursache bzw. auf eine mögliche Konfliktbereinigung gibt.

Beispiel 7.11: Um das Einfügen eines konfliktären Operators zu veranschaulichen, wird Beispiel 7.10 fortgeführt:

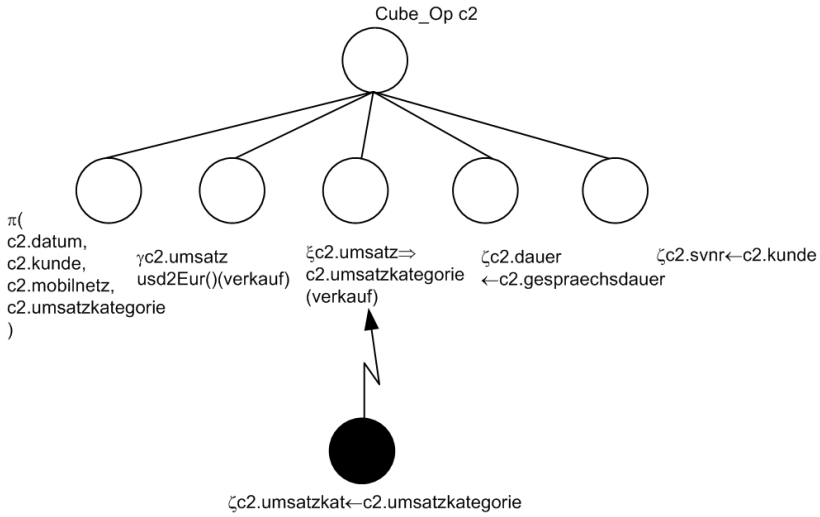


Abbildung 7.16: Operatortree: Einfügen eines konfliktären Operators

Die `checkQueryContext()`-Implementierung des ζ Operators erkennt, dass das umzubenennende dimensionale Attribut im bereits eingefügten ξ Operator verwendet wird. Da ein durch das „Pivoting“ mit dem ξ Operator entferntes dimensionales Attribut nicht umbenannt werden kann, wird eine Fehlermeldung vom Typ `PivotSplitMeasureContextException` erzeugt:

PIVOT MEASURE clause defined as
source-measure: mfuB.verkauf.umsatz
source-dimension: mfuB.umsatzkategorie.umsatzkategorie
could not be processed!
You may have intended to rename a dimensional attribute which is used in a PIVOT MEASURE clause ...

(vgl. Konflikt 6 im Abschnitt 5.3.4.3)

7.5.4 Klasse `SemanticProcessingVisitor`

Die Klasse `SemanticProcessingVisitor` ist die zentrale Klasse in der Architektur des SQL-MDi Query Parser. Sie verwendet die Komponenten „Metadata Management“ und „Operatortree“ um die Baumstruktur mit den DA/FA-Operatoren schrittweise zu konstruieren. Bevor demonstriert wird, wie in der Klasse `SemanticProcessingVisitor` ein DA/FA-Operator erzeugt und in die Baumstruktur eingefügt wird, ist es erforderlich das dieser Klasse zugrunde liegende Visitor-Design Pattern [GHJV95] näher zu erläutern.

7.5.4.1 Visitor-Design Pattern

Mit dem Visitor-Design Pattern können Operationen auf eine Datenstruktur von dieser getrennt werden, wodurch neue Operationen hinzugefügt werden können ohne die Datenstruktur ändern zu müssen [GHJV95].

Jede Element-Klasse der Datenstruktur besitzt eine `accept()` Methode, welche ein Objekt vom Typ `Visitor` als Parameter akzeptiert. `Visitor` ist ein Interface, welches für jede Elementklasse eine `visit()` Methode deklariert, d.h. die `visit()` Methode wird überladen. Die `accept()` Methode einer Elementklasse ruft ihre `visit()` Methode auf, wodurch für jeden Elementtyp der Datenstruktur unterschiedliche Operationen ausgeführt werden können. [GHJV95]

Im Query Parser wird das Visitor-Design Pattern verwendet um aus dem AST eine Operator-Baumstruktur mit den DA/FA-Operatoren zu erzeugen. Alle AST-Klassen besitzen die `accept()` Methode und für alle AST-Klassen existiert eine eigene `visit()` Methode in der Klasse `SemanticProcessingVisitor`. Der dadurch gesteigerte Grad an Flexibilität ermöglicht die Erweiterung von SQL-MDi um zusätzliche Anweisungen bzw. die Einführung von neuen DA/FA-Operatoren ohne Änderungen an den bestehenden Baumknoten-Klassen vornehmen zu müssen (vgl. nicht-funktionale Anforderungen 1 und 3 im Abschnitt 7.1.2). Soll beispielsweise eine neue SQL-MDi-Anweisung hinzugefügt werden, so müssen nur eine entsprechende Klasse für den AST mit einer `accept()` Methode und eine zusätzliche `visit()` Methode in der Klasse `SemanticProcessingVisitor` deklariert werden. Die Erfüllung der nicht-funktionalen Anforderungen „Änderbarkeit“ und „Erweiterbarkeit“ des SQL-MDi Query Parser wird im Abschnitt 7.6 vertieft behandelt.

7.5.4.2 Erzeugen und Einfügen eines DA/FA-Operators

Nachdem im vorherigen Abschnitt das Visitor-Design Pattern erläutert wurde, wird in diesem Abschnitt das Erzeugen und Einfügen eines DA/FA-Operators in die Baumstruktur anhand eines Beispiels präsentiert. Zuvor ist jedoch ein für das Verständnis der Architektur des SQL-MDi Query Parser wichtiger Zusammenhang nochmals zu verdeutlichen:

Jeder SQL-MDi-Anweisung ist ein AST-Knotentyp zugeordnet, wobei für jeden AST-Knotentyp eine `visit()` Methode in der Klasse `SemanticProcessingVisitor` existiert und mit jeder `visit()` Methode ein oder mehrere Strukturknoten bzw. DA/FA-Operatoren erzeugt und in die Baumstruktur eingefügt werden.

Listing 7.8 zeigt die Implementierung einer `visit()` Methode, welche einen oder mehrere α Operatoren (vgl. Abschnitt 5.3.3.1) aufgrund einer `ROLLUP` Anweisung erzeugt und in die Baumstruktur einfügt:

```
1  /*
2  * Erzeugt einen oder mehrere Delete level Operatoren und fügt sie in die
3  * Baumstruktur ein.
4  */
5  public Object visit(ASTRollup node, Object data)
6    // 1. Metadaten ermitteln
7    Token dimName = node.getToDimName();
8    Token cubeAlias = node.getCubeAlias();
9    Token toLevel = node.getToLevel();
10
11   // 2. Kontextbedingungen prüfen
12   // cubealias = alias von ASTCube
13   if(!cubeAlias.image.equals(((ASTCube)node.jjtGetParent()).getAlias().image)){
14     throw new InvalidCubeAliasException(cubeAlias);
15   }
16
17   // 3. Vaterknoten ermitteln
18   Cube_Op cOp = OperatorTreeTool.getAssociatedCubeOp(cubeAlias);
19   String dwNode = cOp.getCube().getPath().getNode();
20   String cubeName = cOp.getCube().getName();
21
22   // 4. Metadaten-Objekte für Parameter erzeugen
23   LevelManager levelMan = new LevelManager(node.getToLevel(),dwNode,cubeName,dimName);
24   Level targetLevel = levelMan.createMetaData(true);
25   // Prüfe, ob targetLevel gelöscht wurde
26   if(OperatorTreeContextTool.rollupTargetIsDeleted(cOp, targetLevel)){
27     QueryParser.addContextException(new RollupContextException(...));
28   }
29   List<Level> levelList = new ArrayList<Level>();
30   levelList.add(targetLevel);
31
32   // Level-Objekte für Delete level Operatoren erzeugen
33                                  // Repository lookup
34   List<Level> levelsToDelete = LevelManager
35                               .getLevelsToDelete(levelList,true);
36   for(Level l : levelsToDelete){
37
38     // 5. Operator-Objekt (z.B. Delete level Operator) erzeugen
39     DA_DeleteLevel_Op delLOp = new DA_DeleteLevel_Op(cOp);
40     delLOp.setLevelToDelete(l);
41
42     // 6. Operator in Baumstruktur einfügen
43     try {
44       cOp.setChild(delLOp.getInsertPosition(), delLOp);
45     } catch (QueryContextException e) {
46       QueryParser.addContextException(e);
47     }
48     // zum Operator-Cache hinzufügen
49     cache.new CachedOperator((LevelPath)l.getPath(),delLOp,cOp);
50   }
51   return null;
52 }
```

Listing 7.8: SemanticProcessingVisitor: Erzeugen und Einfügen von α Operatoren

Die `visit()` Methode in Listing 7.8 akzeptiert als Parameter den AST-Knoten `node` vom Typ `ASTRollup`. Für das Erzeugen und Einfügen eines oder mehrerer α Operatoren aufgrund einer `ROLLUP` Anweisung sind folgende Schritte erforderlich:

1. Metadaten ermitteln (Zeilen 7 - 9):
   Die Werte der Instanzvariablen, welche die für diesen Operator benötigten Metadaten enthalten, werden mit den entsprechenden `get`-Methoden abgefragt.

2. Kontextbedingungen prüfen (Zeilen 13 - 15):
   Für `cubeAlias` ist zu prüfen, ob es dem im Vaterknoten (`node.jjtGetParent()` vom Typ `ASTCube`) deklarierten Alias entspricht. Ist dies nicht der Fall, wird eine Exception vom Typ `InvalidCubeAliasException` generiert, wodurch die Verarbeitung durch den Query Parser gestoppt wird. Der Grund dafür liegt darin, dass ein ungültiger Alias eine ungültige Würfel-Referenz darstellt und dadurch die Baumstruktur nicht korrekt aufgebaut werden kann.

3. Vaterknoten ermitteln (Zeilen 18 - 20):
   Jener Vaterknoten, als dessen Kind ein α Operator eingefügt werden soll, wird mit `OperatorTreeTool.getAssociatedCubeOp(cubeAlias)` ermittelt. Dieser Methodenaufruf liefert jenen Würfel-Strukturknoten vom Typ `Cube_Op`, `cOp`, dessen Alias mit dem übergebenen Parameter übereinstimmt. Wird ein Würfel-Strukturknoten gefunden, werden dessen Metadaten abgefragt, welche für die Initialisierung der α Operatoren verwendet werden; wird kein Würfel-Strukturknoten gefunden, so generiert die Methode `getAssociatedCubeOp(...)` eine Exception vom Typ `InvalidCubeAliasException`.

4. Metadaten-Objekte für Parameter erzeugen (Zeilen 23 - 35):
   Ein α Operator enthält als Parameter die zu löschende Klassifikationsstufe. Die zu löschenden Klassifikationsstufen aufgrund einer `ROLLUP` Anweisung werden durch die mit `TO LEVEL` spezifizierte, neue Klassifikationsstufe feinster Granularität (Basis-Klassifikationsstufe) bestimmt. Mit den Anweisungen der Zeilen 23 - 24 wird aus den dieser Klassifikationsstufe zugeordneten Symbolen (vom Typ `Token`), für den Namen und den Lokalisierungspfad, ein Objekt vom Typ `Level` erzeugt, wobei im Zuge des Aufrufs `levelMan.createMetaData(true)` die Korrektheit des Namens und des Lokalisierungspfads, d.h. des Metadatums, geprüft wird. Handelt es sich um ein ungültiges Metadatum, so generiert die Methode `createMetaData(...)` eine Exception vom Typ `LevelException`. Durch den Methodenaufruf `LevelManager.getLevelsToDelete()` werden die zu löschenden Klassifikationsstufen aufgrund der `ROLLUP` Anweisung ermittelt.

5. Operator-Objekt erzeugen (Zeilen 39 - 40):
   In der `for`-Schleife wird für jede zu löschende Klassifikationsstufe ein α Operator erzeugt, indem bei der Initialisierung `cOp` als Vaterknoten und das aktuelle `Level`-Objekt `l` als Parameter gesetzt wird.

6. Operator in Baumstruktur einfügen (Zeilen 43 - 49):
   Im letzten Schritt wird der α Operator mit der Anweisung `cOp.setChild(delLOp.getInsertPosition(), delLOp)` als Kindknoten des Knotens `cOp` in die Baumstruktur eingefügt. Der α Operator ermittelt dabei durch Aufruf der Methode `getInsertPosition()` selbst seine optimale Einfügeposition. Im Zuge der Abarbeitung dieser Methode werden etwaige Konflikte zu bereits eingefügten Operatoren geprüft und ggf. eine Exception vom Typ `QueryContextException` bzw. eine Konflikt-spezifische `RollupContextException` generiert.

   Da die `ROLLUP` Anweisung einer `DEFINE CUBE` Anweisung zugeordnet ist, werden der(die) erzeugte(n) α Operatore(n) als Kind(er) eines Würfel-Strukturknotens eingefügt. Dieser Zustand ist allerdings nur temporär, da der α Operator (als DA-Operator) eigent-

lich einer Dimension zugeordnet ist. Aus diesem Grund wird ein solcher Operator mit seinen Knoten-Referenzen mit der Variable `cache` vom Typ `OperatorCache` (Zeile 49) zwischengespeichert. Sobald jene `MERGE DIMENSIONS` Anweisung verarbeitet wird, welche die für den zwischengespeicherten DA-Operator relevante Dimension umfasst, wird der DA-Operator als Kind dieses Dimensions-Strukturknotens, umgehängt. Präziser ausgedrückt, es wird die Einfügeposition des DA-Operators in der Kindliste des Dimensions-Strukturknotens neu bestimmt.

Die weiteren `visit()` Methoden der Klasse `SemanticProcessingVisitor` erzeugen die Strukturknoten bzw. DA/FA-Operatoren auf ähnliche Weise.

Die auf den bisherigen Beschreibungen basierende Implementierung des SQL-MDi Query Parser ermöglicht die Erzeugung einer Operator-Baumstruktur aus einer SQL-MDi-Abfrage. Anhang B.2 zeigt die Ausgabe einer vom SQL-MDi Query Parser generierten Operator-Baumstruktur, welche der SQL-MDi-Abfrage des Listing B.1 entspricht.

7.6 Systemänderungen und -erweiterungen

In diesem Abschnitt wird beispielhaft demonstriert, wie Änderungen und Erweiterungen im SQL-MDi Query Parser implementiert werden können. Es soll insbesondere verdeutlicht werden, dass Änderungen und Erweiterungen nur geringe bzw. keine Änderungen an bestehendem Quellcode erfordern. Für die Präsentation der Implementierung einer Änderung bzw. Erweiterung im Query Parser werden folgende Szenarien angenommen:

1. Die SQL-MDi-Syntax soll dahingehend geändert werden, dass eine `ROLLUP` Anweisung nicht mehr einer `DEFINE CUBE` Anweisung, sondern einer `MERGE DIMENSIONS` Anweisung zugeordnet ist (vgl. Anhang A).
2. Die SQL-MDi-Syntax soll um eine Anweisung und die FA um einen neuen Operator für das Hinzufügen eines neuen Kenngrößen-Attributs zu einer Fakttabelle erweitert werden.

7.6.1 Beispiel: Änderung der SQL-MDi-Syntax

Das Verschieben der `ROLLUP` Anweisung von `DEFINE CUBE` zu `MERGE DIMENSIONS` bewirkt folgende Änderungen in der SQL-MDi-Syntax (vgl. Anhang A):

1. Der Aufruf der Regel <`rollup-instr`> in <`source-instance-defs`> wird entfernt.
2. Der Aufruf der Regel <`rollup-instr`> wird in <`merge-dim`> hinzugefügt.

Um diese Syntaxänderung im SQL-MDi Query Parser zu implementieren sind folgende Änderungen vorzunehmen:

1. Die Änderungen in den Regeln <source-instance-defs> und <merge-dim> müssen in den entsprechenden Regeln der SQL-MDi-Grammatik-Spezifikation für JavaCC nachgezogen werden. Da die erforderlichen Änderungen ähnlich der im Anhang A sind, wird hier auf eine Darstellung verzichtet. Nachdem die Änderungen vorgenommen wurden, muss die Grammatik-Spezifikation vom JavaCC-Compiler neu übersetzt werden um die Komponente „Parser“ zu aktualisieren.
2. Die `visit()` Methode für den zugeordneten AST-Knotentyp `ASTRollup`, `visit(ASTRollup node, Object data)`, muss geändert werden (vgl. Abschnitt 7.5.4.2).
   a) Durch die Verschiebung der `ROLLUP` Anweisung ändert sich der Vaterknoten des `ASTRollup`-Knotens. Dieser ist nun nicht mehr Kindknoten eines `ASTCube`-Knotens sondern eines `ASTDimension`-Knotens, d.h. der Schritt 2 in der Verarbeitung eines AST-Knotens, „Kontextbedingungen prüfen“, muss entsprechend geändert bzw. entfernt werden.
   b) Da ein zu erzeugender α Operator nun direkt als Kind eines Dimensions-Strukturknotens eingefügt werden kann, muss der Schritt 3 „Vaterknoten ermitteln“ so geändert werden, dass statt eines Knotens vom Typ `Cube_Op` ein Knoten vom Typ `Dimension_Op` als Vaterknoten ermittelt wird. Daher ist im Schritt 6 „Operator-Objekt erzeugen“ der `Dimension_Op`-Knoten als Vaterknoten des α Operators zu setzen und der Schritt 7 „Operator in Baumstruktur einfügen“ dahingehend zu ändern, dass der α Operator als Kindknoten dieses `Dimension_Op`-Knotens eingefügt wird. Das Zwischenspeichern des Operators mit der Variable `cache` kann entfallen (Zeile 49).

Wie aus diesem Beispiel hervorgeht, sind trotz dieser Modifizierung der SQL-MDi-Syntax, nur Änderungen an zwei Stellen des Query Parser erforderlich: (1) Die Syntaxänderung muss in der JavaCC-Grammatik-Spezifikation berücksichtigt werden. (2) Es müssen Änderungen in der `visit()` Methode für den `ASTRollup`-Knoten vorgenommen werden.

7.6.2 Beispiel: Erweiterung von SQL-MDi und der FA

Die SQL-MDi-Syntax soll um eine neue Anweisung zur Einführung eines neuen Kenngrößen-Attributs erweitert werden (`NEW MEASURE`), welche einer `DEFINE CUBE` Anweisung zugeordnet werden soll. Dieser neuen Anweisung soll ein entsprechender Operator in der FA, Add measure: $\nu_M(c)$, zugewiesen werden, welcher als Parameter das neue Kenngrößen-Attribut M enthält. Dafür sind folgende Erweiterungen im SQL-MDi Query Parser zu implementieren:

1. Defintion einer Produktionsregel für die JavaCC-Grammatik:

```
1   void new_measure() : #NewMeasure:
2   {
3     Token t;
4   }
5   {
6     <NEW_MEASURE>
7     t=<IDENTIFIER> {jjtThis.setCubeAlias(t);}
8     "."
9     t=<IDENTIFIER> {jjtThis.setMeasureAttr(t);}
10  }
```

Listing 7.9: SQL-MDi Erweiterung: Regel für `NEW MEASURE` Anweisung

Listing 7.9 zeigt die neue Produktionsregel für die `NEW MEASURE` Anweisung, welche als Metadaten den Alias des Würfels, für den das neue Kenngrößen-Attribut eingeführt werden soll, sowie den Namen des neuen Kenngrößen-Attributs enthält. Mit der Anweisung `#NewMeasure` wird der Regel ein AST-Knoten vom Typ `ASTNewMeasure` zugeordnet.

2. Deklaration des AST-Knotens `ASTNewMeasure`:
   Für den der Grammatik-Regel aus Listing 7.9 zugeordneten AST-Knoten ist eine Klasse `ASTNewMeasure` zu deklarieren. Es sind als Instanzvariablen `cubeAlias` und `measureAttr` mit den zugehörigen `get/set`-Methoden vorzusehen. Darüber hinaus muss eine `accept()` Methode für das Visitor-Design Pattern (vgl. Abschnitt 7.5.4.1) implementiert werden. Danach kann die erweiterte JavaCC-Grammatik-Spezifikation mit dem JavaCC-Compiler neu übersetzt werden, um die Komponente „Parser“ mit der neuen Regel zu aktualisieren.

3. Deklaration einer Klasse für den FA-Operator $\nu_M(c)$:
   Für den ν Operator ist eine Klasse `FA_AddMeasure_Op` zu deklarieren, welche eine Instanzvariable für das neue Kenngrößen-Attribut vom Typ `Measure` sowie die entsprechenden `get/set`-Methoden enthält.

4. Implementierung der `visit()` Methode für den Knoten `ASTNewMeasure` in der Klasse `SemantikProcessingVisitor`:

```
1   public Object visit(ASTNewMeasure node, Object data){
2     // 1. Metadaten ermitteln
3     Token cubeAlias = node.getCubeAlias();
4     Token measureAttr = node.getMeasureAttr();
5
6     // 2. Kontextbedingungen prüfen
7     if(!cubeAlias.image.equals(((ASTCube)node.jjtGetParent()).getAlias().image)){
8       throw new InvalidCubeAliasException(cubeAlias);
9     }
10
11    // 3. Vaterknoten ermitteln
12    Cube_Op cOp = OperatorTreeTool.getAssociatedCubeOp(cubeAlias);
13    String dwNode = cOp.getCube().getPath().getNode();
14    String cubeName = cOp.getCube().getName();
15
16    // 4. Metadaten-Objekte für Parameter erzeugen
17    Measure newMeasureAttr = new Measure(measureAttr.image,
18                                         new MeasurePath(dwNode,cubeName));
19
20    // 5. Operator-Objekt erzeugen
21    FA_AddMeasure_Op amOp = new FA_AddMeasure_Op(cOp);
22    amOp.setNewMeasure(newMeasureAttr);
23
24    // 6. Operator in Baumstruktur einfügen
25    try {
26      cOp.setChild(amOp.getInsertPosition(), amOp);
27    } catch (QueryContextException e) {
28      QueryParser.addContextException(e);
29    }
30    return null;
31  }
```

Listing 7.10: SQL-MDi Erweiterung: `visit()` Methode für `ASTNewMeasure`

Die `visit()` Methode des Listing 7.10 verwendet die im Abschnitt 7.5.4.2 präsentierten Schritte zum Erzeugen und Einfügen eines Operators in die Baumstruktur, weshalb auf eine weitere Erläuterung verzichtet wird.

Wie aus diesem Beispiel hervorgeht, erfordert die Erweiterung von SQL-MDi um eine neue Anweisung bzw. die FA um einen neuen Operator keine Änderungen an bestehendem Code. Die Implementierung der Beispiel-Erweiterung erforderte nur vier Schritte: (1) Erweiterung der JavaCC-Grammatik um die neue Anweisung, (2) Deklaration einer neuen AST-Knoten-Klasse, (3) Deklaration einer Klasse für den neuen FA-Operator und (4) Implementierung der `visit()` Methode für den neuen AST-Knotentyp zum Erzeugen und Einfügen des Operator-Objekts.

7.7 Zusammenfassung

Aufbauend auf der für die Entwicklung einer Parser-Komponente für SQL-MDi relevanten Theorie der Kapitel 5 und 6 wurde in diesem Kapitel der entwickelte SQL-MDi Query Parser präsentiert. Neben der Erläuterung der Integration des Query Parser in die Architektur eines FDWS und den verwendeten Technologien und Standards, wurde im Zuge der Beschreibung der Systemarchitektur und -implementierung gezeigt, wie die Erfüllung der im Abschnitt 7.1 dokumentierten funktionalen und nicht-funktionalen Anforderungen sichergestellt wird.

Die Systemarchitektur besteht aus drei logischen Komponenten, nämlich der Komponente „Parser“, welche mit Hilfe des Übersetzergenerators JavaCC generiert wurde und der Syntaxanalyse dient, der Komponente „Metadata Management“, welche für das Prüfen und Erzeugen von Metadaten-Objekten verantwortlich ist, und der Komponente „Operatortree“, welche die Baumstruktur mit den DA/FA-Operatoren repräsentiert. Die Brücke zwischen diesen Komponenten bildet die Klasse `SemanticProcessingVisitor`, welche eine Implementierung des Visitor-Design Patterns [GHJV95] darstellt und für den Aufbau der Baumstruktur zuständig ist. Der wesentliche Vorteil der Berücksichtigung des Visitor-Design Patterns [GHJV95] bei der Implementierung des SQL-MDi Query Parser ist die erhöhte Flexibilität bezüglich möglicher Änderungen und Erweiterungen. Beispielhaft wurde im Abschnitt 7.6 demonstriert, dass durch das gewählte Design eine hohe Erweiterbarkeit und Änderbarkeit erreicht wird.

Kapitel 8

Evaluierung, Resumee und Ausblick

Im folgenden Abschnitt wird zunächst demonstriert, wie die Baumstruktur mit den DA/FA-Operatoren verarbeitet werden kann um ein integriertes Ergebnis, d.h. einen globalen, instanziierten Würfel, zu erzeugen. Die Baumstruktur wird also hinsichtlich der Verarbeitbarkeit durch eine Prozessor-Komponente evaluiert. Die zentralen Aussagen der vorliegenden Diplomarbeit werden im Abschnitt 8.2 nochmals hervorgehoben, bevor im Abschnitt 8.3 ein Ausblick auf mögliche Verbesserungs- und Erweiterungsmöglichkeiten des SQL-MDi Query Parser gegeben wird.

8.1 Evaluierung

Wie bereits im Kapitel 1 erwähnt, stellt der entwickelte SQL-MDi Query Parser eine Teil-Komponente eines Abfrage-Werkzeugs in einem Föderierten Data Warehouse-System (FDWS) dar. Um sicherzustellen, dass aus der vom Query Parser erzeugten Operator-Baumstruktur ein integriertes Ergebnis, d.h. ein globaler, instanziierter Würfel, berechnet werden kann, ist ein Integrationstest [ZR06] erforderlich. Da sich die die Baumstruktur verarbeitende Prozessor-Komponente [Ros08] zum relevanten Zeitpunkt noch in einem frühen Entwicklungsstadium befand, wurde deren Funktionalität auf primitive Weise simuliert um eine Evaluierung der Operator-Baumstruktur zu ermöglichen.

Um den Aufwand für die Implementierung der Prozessor-Simulation möglichst gering zu halten, wurden folgende Annahmen und Vereinfachungen berücksichtigt:

- Die Komponenten-DWS werden durch Datenstrukturen in Java simuliert, wodurch von Interaktionen mit realen Data Warehouses abstrahiert werden kann (vgl. Abbildung 8.1).
- Für den Integrationstest werden mit Star-Schemata modellierte Würfel in den Komponenten-DWS angenommen (vgl. nicht-funktionale Anforderung 6 im Abschnitt 7.1.2).
- Für alle Attribute der Star-Schemata wird der Datentyp „String“ angenommen.
- Die DA/FA-Operatoren modifizieren die Schemata und Instanzen der simulierten Komponenten-DWS.

- Bei der Verarbeitung der Baumstruktur werden Effizienz- und Geschwindigkeits-Gesichtspunkte nicht berücksichtigt.
- Die Implementierung der Simulation erfolgte mit dem Ziel, dass aus der dem Listing B.1 entsprechenden Baumstruktur ein integriertes Ergebnis erzeugt werden kann. Aus diesem Grund wurden auch nicht alle DA/FA-Operatoren vollständig implementiert; so ist z.B. nur die Aggregatsfunktion SUM verfügbar.

8.1.1 Design des Integrationstests

Abbildung 8.1 visualisiert die für die Simulation der Prozessor-Komponente entworfenen Klassen:

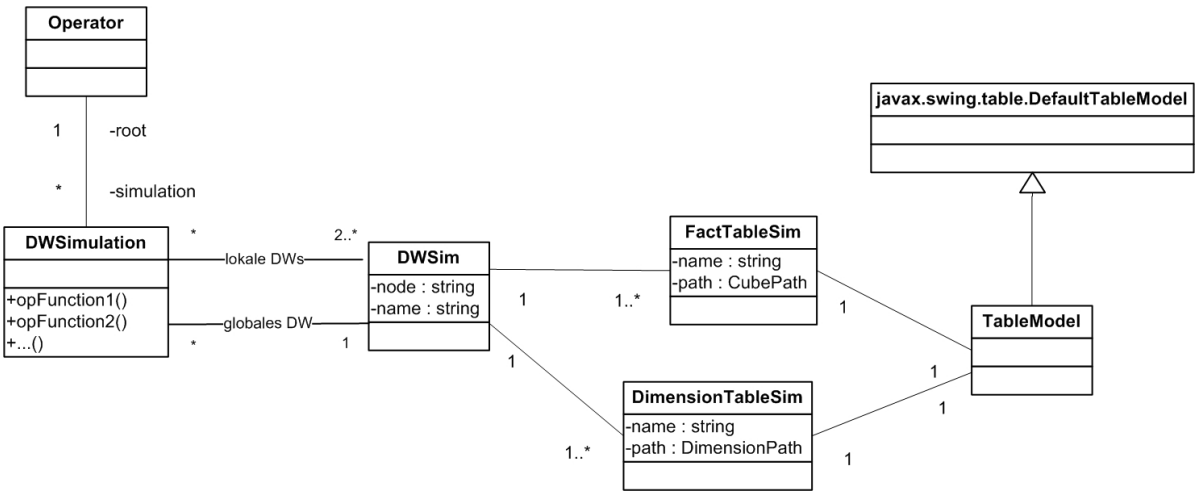


Abbildung 8.1: Simulation des Query Prozessors

Ein lokales bzw. das globale Data Warehouse wird durch ein Objekt der Klasse `DWSim` repräsentiert. Ein `DWSim`-Objekt wird durch den DW-Knoten `node` und den DW-Namen `name` identifiziert und umfasst ein oder mehrere Fakt- und Dimensionstabellen. Die Fakt- und Dimensionstabellen werden durch Objekte der Klassen `FactTableSim` bzw. `DimensionTableSim` repräsentiert, welche neben dem Namen `name` und einer Pfadangabe `path` ein `TableModel`-Objekt referenzieren.

`TableModel` stellt einen von `javax.swing.table.DefaultTableModel` abgeleiteten, erweiterten Tabellentyp dar, welcher für die Modellierung von Star-Schemata verwendet wird. Die Implementierung der Funktionalität der DA/FA-Operatoren befindet sich in der Klasse `DWSimulation`, in welcher die Operator-Baumstruktur traversiert wird, die lokalen Würfel mit den DA/FA-Operatoren modifiziert werden und der globale, instanziierte Würfel berechnet wird.

8.1.2 Ergebnis des Integrationstests

Anhang B dokumentiert den durchgeführten Integrationstest. Den Input für den SQL-MDi Query Parser stellt die SQL-MDi-Abfrage des Listing B.1 dar, welche die Schema- und Instanzheterogenitäten der Abbildung 3.4 bzw. der Tabellen 3.2 und 3.3 eliminiert. Aus dieser Abfrage generiert der SQL-MDi Query Parser eine Operator-Baumstruktur (vgl. Abschnitt B.2), welche von der simulierten Prozessor-Komponente verarbeitet wurde. Im Zuge der Verarbeitung wurden die (nummerierten) Konflikte zwischen den Schemata und Instanzen der lokalen Data Warehouses „local data warehouse 1“ und „local data warehouse 2“ durch Anwendung der DA/FA-Operatoren aufgelöst und „global data warehouse“, d.h. der globale, instanziierte Würfel, erzeugt (vgl. Abschnitt B.2).

Trotz der getroffenen Annahmen und vorgenommenen Vereinfachungen veranschaulicht dieser Integrationstest, dass aus einer vom SQL-MDi Query Parser erzeugten Operator-Baumstruktur ein integriertes Ergebnis, d.h. ein globaler, instanziierter Würfel, berechnet werden kann.

8.2 Resumee

Die Integration von unabhängigen DBS und DWS wird vor allem durch bestehende Heterogenitäten auf Schema- und Instanzebene erschwert. Für die Integration von unabhängigen DWS wurde ein Architekturmodell für ein FDWS präsentiert, das transparenten Zugriff auf die Komponenten-DWS für die Benutzer gewährleistet. Zur Auflösung der zwischen den Komponenten-DWS bestehenden Heterogenitäten ist die Verwendung einer speziellen Abfragesprache in der Föderations-Schicht des FDWS vorgesehen.

Bei der Integration von DWS können, im Vergleich zur Integration von DBS, aufgrund des semantisch reichhaltigeren, multidimensionalen Datenmodells zusätzliche Heterogenitäten auftreten. Deshalb eignen sich die untersuchten Abfragesprachen für MDBS, welche für relationale DBS entwickelt wurden, nicht für deren Beseitigung. Aus diesem Grund wurde die multidimensionale Abfragesprache SQL-MDi speziell für den Einsatz im vorgestellten FDWS konzipiert.

SQL-MDi unterstützt die Auflösung von Heterogenitäten auf Schema- und Instanzebene und ermöglicht die Erzeugung eines globalen, instanziierten Würfels. Wie für SQL, existiert auch für SQL-MDi eine formale Algebra, nämlich die Dimension-Algebra (DA) und Fact-Algebra (FA). Die DA/FA umfasst Operatoren, welche auf Dimensionen (DA) und Fakten (FA) angewendet werden, um bestehende Heterogenitäten aufzulösen. Aufgrund des zugrunde liegenden kanonischen, multidimensionalen Datenmodells kann SQL-MDi bzw. die DA/FA für die Integration von DW-Produkten unterschiedlicher Hersteller verwendet werden.

Für die im FDWS verwendete Abfragesprache SQL-MDi wurde im Zuge dieser Diplomarbeit eine Parser-Komponente, der SQL-MDi Query Parser, entwickelt. Die Aufgabe des Query Parser ist die Analyse einer SQL-MDi-Abfrage und deren Transformation in eine Operator-Baumstruktur, welche die den SQL-MDi-Anweisungen entsprechenden DA/FA-Operatoren

strukturiert. Eine SQL-MDi-Abfrage wird hinsichtlich der Korrektheit der Syntax, der Korrektheit der enthaltenen Metadaten sowie hinsichtlich logischer Konflikte, welche zwischen Operatoren bestehen können, analysiert. Die Prüfung der Korrektheit der enthaltenen Metadaten erfolgt durch Abgleich mit dem Metadaten-Repository, welches Informationen über die Komponentenschemata speichert. Die Erkennung logischer Konflikte zwischen Operatoren wurde derart realisiert, dass für einen einzufügenden Operator zunächst geprüft wird, ob er mit einem bereits eingefügten Operator einen Konflikt erzeugen würde. Über Syntax- und Metadaten-Fehler bzw. über erkannte logische Konflikte wird der Benutzer mit aussagekräftigen Fehlermeldungen informiert.

Bei der Erzeugung der Operator-Baumstruktur durch den SQL-MDi Query Parser wird sichergestellt, dass diese durch eine Prozessor-Komponente, ohne Umstrukturierungen vorzunehmen, verarbeitet werden und ein integriertes Ergebnis berechnet werden kann. Aus diesem Grund implementiert der Query Parser spezielle Einfüge-Regeln für voneinander abhängige Operatoren, sodass bestimmte Operatoren vor bestimmten anderen Operatoren einzufügen sind, wenn definierte Bedingungen erfüllt sind. Darüber hinaus erfolgt der Aufbau der Baumstruktur unter Berücksichtigung von Optimierungen auf logischer Ebene, indem Operatoren, welche die Ergebnismenge verkleinern, so eingefügt werden, dass sie von einer Prozessor-Komponente zuerst verarbeitet werden.

Da der SQL-MDi Query Parser primär als Prototyp für die Evaluierung der Abfragesprache SQL-MDi und der DA/FA entwickelt wurde, zeichnet sich dieser durch leichte Änderbarkeit und Erweiterbarkeit aus. Dadurch ist es möglich Änderungen und Erweiterungen an der SQL-MDi-Syntax und an der DA/FA mit geringen bzw. keinen Änderungen an bestehendem Quellcode zu implementieren.

Die Evaluierung des SQL-MDi Query Parser bzw. der Verarbeitbarkeit der erzeugten Baumstruktur zeigte, dass SQL-MDi sowie die mit Hilfe eines Baumes strukturierten Operatoren der DA/FA geeignet sind um bestehende Schema- und Instanzheterogenitäten aufzulösen und ein integriertes Ergebnis, d.h. einen globalen, instanziierten Würfel, zu berechnen.

8.3 Ausblick

In diesem Abschnitt werden die im Rahmen dieser Diplomarbeit nicht behandelten Aspekte sowie diverse Verbesserungs- und Erweiterungsmöglichkeiten für den SQL-MDi Query Parser erläutert.

Die Evaluierung der Eignung von SQL-MDi für die Integration von DWS, welche kein Star-Schema als logisches Modell implementieren, war nicht Teil dieser Diplomarbeit. Deshalb wurde der entwickelte SQL-MDi Query Parser nicht dahingehend evaluiert, ob beispielsweise auch MOLAP-Systeme oder Systeme mit Snowflake-Schemata integriert werden können. Eine Integration von DWS mit unterschiedlichen logischen Modellen erscheint jedoch als durchaus relevant.

Verbesserungs- bzw. Erweiterungsmöglichkeiten des SQL-MDi Query Parser bestehen vor allem im Falle von erkannten Konflikten zwischen Operatoren. In der aktuellen Implementierung wird der Benutzer beim Erkennen eines Konflikts mit einer entsprechenden Fehlermeldung darüber in Kenntnis gesetzt. Wünschenswert wäre jedoch ein „intelligenterer“ Parser, welcher beim Erkennen eines Konflikt selbst versucht diesen aufzulösen. Daher sind die zwischen Operatoren der DA/FA möglichen Konflikte dahingehend zu analysieren, ob eine entsprechende Regel zur Konfliktauflösung definiert werden kann. Diese Regeln könnten dann die bisher produzierten Fehlermeldungen ersetzen.

Literaturverzeichnis

[ABJ+03] Akinde, M.O., M.H. Böhlen, T. Johnson, Laks V.S. Lakshmanan und D. Srivastava: *Efficient OLAP query processing in distributed data warehouses.* Inf. Syst., 28(1-2):111–135, 2003.

[AL98] Albrecht, J. und W. Lehner: *On-line Analytical Processing in Distributed Data Warehouses.* In: *IDEAS*, Seiten 78–85, 1998.

[AO83] Albert, J. und T. Ottman: *Automaten, Sprachen und Maschinen für Anwender.* Bibliographisches Institut, 1983.

[App98] Appel, A.W.: *Modern compiler implementation in Java.* Cambridge University Press, 1998.

[ASU88] Aho, A.V., R. Sethi und J.D. Ullman: *Compilerbau.* Addison-Wesley, 1988.

[AU72] Aho, A.V. und J.D. Ullman: *The Theory of Parsing, Translation, and Compiling. Volume I: Parsing.* Prentice Hall Professional Technical Reference, 1972.

[BG04] Bauer, A. und H. Günzel: *Data Warehouse Systeme - Architektur, Entwicklung, Anwendung.* dpunkt.Verlag GmbH, 2. Auflage, 2004.

[BL97] Bauer, A. und W. Lehner: *The Cube-Query-Language(CQL) for Multidimensional statistical and scientific Database Systems.* In: *DASFAA*, Seiten 263–272, 1997.

[Bre90] Breitbart, Y.: *Multidatabase Interoperability.* SIGMOD Record, 19(3):53–60, 1990.

[BS06a] Berger, S. und M. Schrefl: *Analysing Multi-dimensional Data Across Autonomous Data Warehouses.* In: *DaWaK*, Seiten 120–133, 2006.

[BS06b] Berger, S. und M. Schrefl: *Extended Syntax Definition for SQL-MDi. Technical report, available at http://www.dke.jku.at.*, 2006.

[BS08] Berger, S. und M. Schrefl: *From Federated Databases to a Federated Data Warehouse System.* In: *Proceedings of the 41st Hawaii International Conference on System Sciences (HICSS-41 2008)*, 2008.

[CD97] Chaudhuri, S. und U. Dayal: *An Overview of Data Warehousing and OLAP Technology.* SIGMOD Record, 26(1):65–74, 1997.

[CKW93] Chen, W., M. Kifer und D.S. Warren: *HILOG: A Foundation for Higher-Order Logic Programming.* J. Log. Program., 15(3):187–230, 1993.

[Cod70] Codd, E.F.: *A Relational Model of Data for Large Shared Data Banks.* Commun. ACM, 13(6):377–387, 1970.

[Con97] Conrad, S.: *Föderierte Datenbanksysteme - Konzepte der Integration.* Springer Verlag, 1997.

[CT04] Cabibbo, L. und R. Torlone: *On the Integration of Autonomous Data Marts.* In: *SSDBM*, Seiten 223–234, 2004.

[CT05] Cabibbo, L. und R. Torlone: *Integrating Heterogeneous Multidimensional Databases.* In: *SSDBM*, Seiten 205–214, 2005.

[dBGV96] Bussche, J. Van den, D. Van Gucht und G. Vossen: *Reflective Programming in the Relational Algebra.* J. Comput. Syst. Sci., 52(3):537–549, 1996.

[DG06] Dorn, J. und G. Gottlob: *Künstliche Intelligenz.* In: *Informatik Handbuch.* Carl Hanser Verlag, 3. Auflage, 2006.

[DJGM] Dalkilic, M.M., M. Jain, D. Van Gucht und A. Mendhekar: *Design and Implementation of Reflective SQL (Extended Abstract).*

[DP82] DeRemer, F. und T. Pennello: *Efficient Computation of LALR(1) Look-Ahead Sets.* ACM Trans. Program. Lang. Syst., 4(4):615–649, 1982.

[Ecl] *Eclipse - an open development platform.* http://www.eclipse.org/. abgerufen am 09.11.2007.

[GCB+97] Gray, J., S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow und H. Pirahesh: *Data Cube: A Relational Aggregation Operator Generalizing Group-by, Cross-Tab, and Sub Totals.* Data Min. Knowl. Discov., 1(1):29–53, 1997.

[GHJV95] Gamma, E., R. Helm, R. Johnson und R. Vlissides: *Design Patterns - Elements of reusable Object-Oriented Software.* Addison-Wesley, 1995.

[GL98] Gingras, F. und Laks V.S. Lakshmanan: *nD-SQL: A Multi-Dimensional Language for Interoperability and OLAP.* In: *VLDB*, Seiten 134–145, 1998.

[GLRS93] Grant, J., W. Litwin, N. Roussopoulos und T.K. Sellis: *Query Languages for Relational Multidatabases.* VLDB J., 2(2):153–171, 1993.

[GLS96] Gyssens, M., Laks V.S. Lakshmanan und I.N. Subramanian: *Tables as a Paradigm for Querying and Restructuring.* In: *PODS*, Seiten 93–103, 1996.

[GLS+97] Gingras, F., Laks V. S. Lakshmanan, I.N. Subramanian, D. Papoulis und N. Shiri: *Languages for Multi-database Interoperability.* In: *SIGMOD Conference*, Seiten 536–538, 1997.

[GMR98] Golfarelli, M., D. Maio und S. Rizzi: *The Dimensional Fact Model: A Conceptual Model for Data Warehouses.* Int. J. Cooperative Inf. Syst., 7(2-3):215–247, 1998.

[GR98] Golfarelli, M. und S. Rizzi: *Methodological Framework for Data Warehouse Design.* In: *DOLAP*, Seiten 3–9, 1998.

[HGM05] Hurtado, C.A., C. Gutiérrez und A.O. Mendelzon: *Capturing summarizability with integrity constraints in OLAP.* ACM Trans. Database Syst., 30(3):854–886, 2005.

[HKKR05] Hitz, M., G. Kappel, E. Kapsammer und W. Retschitzegger: *UML @ Work.* dpunkt-Verlag, 3. Auflage, 2005.

[HM93] Hammer, J. und D. McLeod: *An Approach to Resolving Semantic Heterogenity in a Federation of Autonomous, Heterogeneous Database Systems.* Int. J. Cooperative Inf. Syst., 2(1):51–83, 1993.

[HMU02] Hopcroft, J.E., R. Motwani und J.D. Ullman: *Einführung in die Automatentheorie, Formale Sprachen und Komplexitätstheorie.* Pearson Studium, 2. Auflage, 2002.

[Inm05] Inmon, W.H.: *Building the Data Warehouse.* Wiley Publishing, Inc., 4th Edition, 2005.

[Java] *Java Database Connectivity.* http://java.sun.com/javase/technologies/database/. abgerufen am 09.11.2007.

[Javb] *Java Eclipse Plugin 1.5.7.* http://perso.orange.fr/eclipse_javacc/. abgerufen am 09.11.2007.

[Javc] *Java.sun.com. The Source for Java Developers.* http://www.java.sun.com. abgerufen am 09.11.2007.

[JMG95] Jain, M., A. Mendhekar und D. Van Gucht: *A Uniform Data Model for Relational Data and Meta-Data Query Processing.* In: *COMAD*, Seiten 1–25, 1995.

[JUn] *JUnit.* http://www.junit.org/index.html. abgerufen am 09.11.2007.

[Kas99] Kastens, U.: *Übersetzerbau.* Oldenbourg, 1999.

[KE04] Kemper, A. und A. Eickler: *Datenbanksysteme - Eine Einführung.* Oldenbourg, 5. Auflage, 2004.

[KLK91] Krishnamurthy, R., W. Litwin und W. Kent: *Language Features for Interoperability of Databases with Schematic Discrepancies.* In: *SIGMOD Conference*, Seiten 40–49, 1991.

[Len02] Lenzerini, M.: *Data integration: a theoretical perspective.* In: *PODS '02: Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, Seiten 233–246, New York, NY, USA, 2002. ACM Press.

[LL03] Levene, M. und G. Loizou: *Why is the snowflake schema a good data warehouse design?* Inf. Syst., 28(3):225–240, 2003.

[LMR90] Litwin, W., L. Mark und N. Roussopoulos: *Interoperability of Multiple Autonomous Databases.* ACM Comput. Surv., 22(3):267–293, 1990.

[LMSS95] Levy, A.Y., A.O. Mendelzon, Y. Sagiv und D. Srivastava: *Answering Queries Using Views.* In: *PODS*, Seiten 95–104, 1995.

[LMW] Löberbauer, M., H. Mössenböck und A. Wöss: *The Compiler Generator Coco/R.* http://www.ssw.uni-linz.ac.at/Research/Projects/Coco/. abgerufen am 09.11.2007.

[Log] *Log4J.* http://logging.apache.org/log4j/docs/index.html. abgerufen am 09.11.2007.

[LRT96] Lehner, W., T. Ruf und M. Teschke: *CROSS-DB: A Feature-Extended Multidimensional Data Model for Statistical and Scientific Databases.* In: *CIKM*, Seiten 253–260, 1996.

[LSS93] Lakshmanan, Laks V.S., F. Sadri und I.N. Subramanian: *On the Logical Foundations of Schema Integration and Evolution in Heterogeneous Database Systems.* In: *DOOD*, Seiten 81–100, 1993.

[LSS97] Lakshmanan, Laks V.S., F. Sadri und I.N. Subramanian: *Logic and Algebraic Languages for Interoperability in Multidatabase Systems.* J. Log. Program., 33(2):101–149, 1997.

[LSS99] Lakshmanan, Laks V.S., F. Sadri und S.N. Subramanian: *On Efficiently Implementing SchemaSQL on an SQL Database System.* In: *VLDB*, Seiten 471–482, 1999.

[LSS01] Lakshmanan, Laks V.S., F. Sadri und S.N. Subramanian: *SchemaSQL: An extension to SQL for multidatabase interoperability.* ACM Trans. Database Syst., 26(4):476–519, 2001.

[MDX] MDX: *Multidimensional Expressions (MDX) Referenz.* http://msdn2.microsoft.com/de-de/library/ms145506.aspx. abgerufen am 09.11.2007.

[Mic] Microsystems, Sun: *Java compiler compiler (JavaCC) - the Java parser generator.* https://javacc.dev.java.net/. abgerufen am 09.11.2007.

[MR95] Missier, P. und M. Rusinkiewicz: *Extending a Multidatabase Manipulation Language to Resolve Schema and Data Conflicts.* In: *DS-6*, Seiten 93–115, 1995.

[Mös06a] Mössenböck, H.: *Übersetzer.* In: *Informatik Handbuch.* Carl Hanser Verlag, 3. Auflage, 2006.

[Mös06b] Mössenböck, H.: *The Compiler Generator Coco/R. User Manual*, 2006.

[MV00] Masermann, U. und G. Vossen: *SISQL: Schema-Independent Database Querying (On and Off the Web).* In: *IDEAS*, Seiten 55–64, 2000.

[OMG01] OMG: *Common Warehouse Metamodel (CWM) Specification*, 2001.

[ÖV99] Özsu, M.T. und P. Valduriez: *Principles of Distributed Database Systems.* Prentice-Hall, 2nd Edition, 1999.

[Par] Parr, T.: *Antlr, another tool for language recognition.* http://www.antlr.org. abgerufen am 09.11.2007.

[PCTM02] Poole, J., D. Chang, D. Tollbert und D. Mellor: *Common Warehouse Metamodel.* John Wiley & Sons Inc, 2002.

[PP04] Pomberger, G. und W. Pree: *Software Engineering. Architektur-Design und Prozessorientierung.* Carl Hanser Verlag, 3. Auflage, 2004.

[PRP02] Pedersen, D., K. Riis und T.B. Pedersen: *A Powerful and SQL-Compatible Data Model and Query Language for OLAP.* In: *Australasian Database Conference*, 2002.

[PS04] Preuner, G. und M. Schrefl: *Herausforderungen der Wirtschaftsinformatik.* In: *Methoden und Modelle für den Entwurf von Data Warehouses.* Deutscher Universitäts-Verlag (DUV), 2004.

[RB01] Rahm, E. und P.A. Bernstein: *A survey of approaches to automatic schema matching.* VLDB J., 10(4):334–350, 2001.

[Rec06] Rechenberg, P.: *Formale Sprachen und Automaten.* In: *Informatik Handbuch.* Carl Hanser Verlag, 3. Auflage, 2006.

[RGW99] Rood, C., D. Van Gucht und F. Wyss: *MD-SQL: A Language for Meta-data Queries over Relational Databases*, 1999.

[Ros08] Rossgatterer, T.: *Enwicklung eines Query Prozessors für ein Föderiertes Data Warehouse-System (in Arbeit befindliche Diplomarbeit)*, 2008.

[Sch08] Schweinzer, P.: *Enwicklung eines visuellen Integrations-Tools für verteilte Data Warehouse Schemata (in Arbeit befindliche Diplomarbeit)*, 2008.

[Ski98] Skiena, S.S.: *The algorithm design manual.* 1998.

[SL90] Sheth, A.P. und J.A. Larson: *Federated Database Systems for Managing Distributed, Heterogeneous, and Autonomous Databases.* ACM Comput. Surv., 22(3):183–236, 1990.

[SQLa] SQL2003: *International Organization for Standardization. ISO/IEC 9075:2003: Title: Information Technology - Database Languages - SQL.*

[SQLb] SQL99: *International Organization for Standardization. ISO/IEC 9075:1999: Title: Information Technology - Database Languages - SQL.*

[TP05] Torlone, R. und I. Panella: *Design and Development of a Tool for Integrating Heterogeneous Data Warehouses.* In: *DaWaK*, Seiten 105–114, 2005.

[Ull97] Ullman, J.D.: *Information Integration Using Logical Views.* In: *ICDT*, Seiten 19–40, 1997.

[vdM98] Meyden, R. van der: *Logical Approaches to Incomplete Information: A Survey.* In: *Logics for Databases and Information Systems*, Seiten 307–356, 1998.

[WG01] Wyss, C.M. und D. Van Gucht: *A Relational Algebra for Data/Metadata Integration in a Federated Database System.* In: *CIKM*, Seiten 65–72, 2001.

[Wir85] Wirth, N.: *Programmieren in Modula-2.* Springer Verlag, 1985.

[WR05] Wyss, C.M. und E.L. Robertson: *Relational languages for metadata integration.* ACM Trans. Database Syst., 30(2):624–660, 2005.

[ZR06] Züllighoven, H. und J. Raasch: *Softwaretechnik.* In: *Informatik Handbuch.* Carl Hanser Verlag, 3. Auflage, 2006.

Anhang A

Spezifikation der SQL-MDi-Syntax

In diesem Anhang folgt eine Dokumentation der SQL-MDi-Syntax in EBNF-Notation [Wir85]. Darüber hinaus erfolgt eine Zuordnung der DA/FA-Operatoren zu den entsprechenden Grammatik-Regeln bzw. Teilen von Grammatik-Regeln mit Kommentaren. In den Kommentaren wird zusätzlich die Multiplizität der Operatoren angegeben, d.h. wie viele Operatoren eines bestimmten Typs aufgrund einer Grammatik-Regel bzw. eines Teils einer Grammatik-Regel erzeugt werden können.

0. Struktur einer SQL-MDi-Abfrage

<sql-mdi-query> ::= <define-clause> <merge-dim-clause> <merge-cubes-clause>.

A. DEFINE (Sub-)Anweisungen

<define-clause> ::= “DEFINE” <cube-specs>.

<cube-specs> ::= <cube-spec> { [“,”] <cube-spec>} [“,”] <global-cube-spec>.

<global-cube-spec> ::= “GLOBAL CUBE” <node> “::” <cube-name> “AS” <global-cube-alias>.

<cube-spec> ::= “CUBE” <node> “::” <cube-name> “AS” <cube-alias>
<source-schema-defs> <source-instance-defs>.

<source-schema-defs> ::= [“(“]<measure-imports> [“,”] <dim-imports> [[“,”] <pivot-schema>] [“)”].

<measure-imports> ::= “MEASURE” <measure-import> { [“,”] <measure-import>}.

<measure-import> ::= <cube-alias> “.” <measure-attr-name> [“->”<new-measure-attr-name>].

<dim-imports> ::= <dim-import> { [“,”] <dim-import>}.

<dim-import> ::= “DIM” <cube-alias> “.” <dim-attr-name> [“->” <new-dim-attr-name>]
[<level-mappings-subclause>].

<level-mappings-subclause>::=[“(“] “MAP LEVELS” <cube-alias> ”.” <dim-name> <mapped-levels-list> [“)”].

<mapped-levels-list> ::= [“(“] “[“ <level-name> [“->” <new-level-name>] “]”
{ [“,”] “[“ <level-name> [“->” <new-level-name>] “]” } [“)”].

<pivot-schema> ::= “PIVOT” (<pivot-split-measure-attr> | <pivot-merge-measure-attrs>).

<pivot-split-measure-attr> ::= “MEASURE” <cube-alias> “.” <measure-attr-name> “BASED ON” <cube-alias> “.”
<dim-attr-name> [<rename-context-dim-insts>].

<rename-context-dim-insts> ::= <rename-context-dim-inst> { [“,”] <rename-context-dim-inst>}.

<rename-context-dim-inst>::= [“(“] “RENAME CONTEXT DIM” <cube-alias> “.” <context-dim-attr-name>
[“‘”] <old-value> [“‘”] “>> “ [“‘”] <new-value> [“‘”] [“)”].

<pivot-merge-measure-attrs> ::= “MEASURES” <measure-attr-list> “INTO” <cube-alias> “. “
<measure-attr-name> “USING” <cube-alias> “.” <context-dim-attr>.

<measure-attr-list> ::= <cube-alias> “.” <measure-attr-name> { [“,”] <cube-alias> “.”
<measure-attr-name>}.

<source-instance-defs> ::= {<rollup-instr>}, {<measure-conversion-instrs>}.

<rollup-instr> ::= [“(“] “ROLLUP” <cube-alias> “.” <dim-attr-name> “TO LEVEL”
<cube_alias> “.” <dim-name>”[“ <level-name> “]” [“)”].

<measure-conversion-instrs> ::= [“(“] “CONVERT MEASURES APPLY” <measure-conversion-instr>
{[“,”]<measure-conversion-instr>} [“)”].

<measure-conversion-instr> ::= <function-name> “FOR” <cube-alias> “.”
<measure-attr-name> (“DEFAULT” | “WHERE” <conditions>).

<conditions> ::= <condition> {<condition>}.

<condition> ::= [“AND” | “OR”] <cube-alias> “.” [<dim-name> “.”] <dim-attr-name> <operator>
(<cube-alias> “.” [<dim-name> “.“] <dim-attr-name> | [“‘”] <value> [“‘”]).

Kommentar [WB1]: * $\lambda_{(N \subset MF)}(c)$

Kommentar [WB2]: 1 $\varsigma_{M' \leftarrow M}(c)$

Kommentar [WB3]: 1 $\pi_{(L \subset Ac)}(c)$

Kommentar [WB4]: 1 $\varsigma_{A' \leftarrow A}(c)$

Kommentar [WB5]: * $\alpha_{ij}(d)$

Kommentar [WB6]: 1 $\varsigma_{I' \leftarrow I}(d)$

Kommentar [WB7]: 1 $\xi_{M' \Rightarrow A'}(c)$

Kommentar [WB8]: 1 $\delta_{w \leftarrow v(l.kj)}(d)$

Kommentar [WB9]: 1 $\chi_{ML \Rightarrow Mx,\, Ax}(c)$

Kommentar [WB10]: 1..* $\alpha_{ij}(d)$

Kommentar [WB11]: 1..* $\gamma_{M' \theta}(c)$

Kommentar [WB12]: 1 $\sigma_{P}(c)$

B. MERGE DIMENSIONS (Sub-)Anweisungen

<merge-dim-clause> ::= <merge-dim> { [“,“] <merge-dim>}.

<merge-dim> ::= [“(“] “MERGE DIMENSIONS” <dim-list> “INTO” <dim-def> [<set-operation>] {member-rel-subclause} {<rename-instance-instrs>} {<attribute-mappings>} [<attr-conversion-instrs>] [“)”].

Kommentar [WB13]: `mergeDim`

<dim-list> ::= <dim-def> { [“,”] <dim-def>}.

<dim-def> ::= (<global-cube-alias> | <cube-alias>) “.” <dim-name> “AS” <dim-alias>.

<member-rel-subclause> ::= [“(“] “RELATE” <levels-list> <join-conditions> “USING HIERARCHY OF” <dim-alias> [“)”].

Kommentar [WB14]: $1\ \Omega_{m \rightarrow v}(d)$

<levels-list> ::= <dim-alias> “.” <level-name> { [“,”] <dim-alias> “.” <level-name>}.

<join-conditions> ::= “WHERE” <join-condition> {“AND” <join-condition>}.

<join-condition> ::= <dim-alias> “.” <primary-key> “=” <dim-alias> “.” <primary-key>.

<rename-instance-instrs> ::= [“(“] “RENAME” <dim-alias> “.” <dim-attr-name> (<mapping-table-spec> | <rename-instance-instr> { [“,”] <rename-instance-instr>}) [“)”].

Kommentar [WB15]: $1..* \ \delta_{w \leftarrow v(1.kj\ |\ 1.ai)}(d)$

<mapping-table-spec> ::= “USING MAPPINGTABLE” <node> “::” <table-name> “TO” <dim-alias> “.” <dim-attr-name>.

<rename-instance-instr> ::= “>>” [“’”] <value> [“’”] [<”WHERE” <conditions>].

<attribute-mappings> ::= [“(“] “MATCH ATTRIBUTES” <attr-def> “IS” <attr-list> [“)”].

Kommentar [WB16]: $1..* \ \varsigma_{l' \leftarrow l}(d)$ bzw. $\varsigma_{a' \leftarrow l.a}(d)$

<attr-list> ::= <attr-def> { [“,”]<attr-def>}.

<attr-def> ::= <dim-alias> “.” <attr-name>.

<attr-conversion-instrs> ::= [“(“] “CONVERT ATTRIBUTES APPLY” <attr-conversion-instr> { [“,”]<attr-conversion-instr>} [“)”].

Kommentar [WB17]: $1..* \ \gamma_{(1.kj\ |\ 1.ai)\ \theta}(d)$

<attr-conversion-instr> ::= <function-name> “FOR” <dim-alias> “.” <dim-attr-name> (“DEFAULT” | “WHERE” <conditions>).

C. MERGE CUBES (Sub-)Anweisungen

<merge-cubes-clause> ::= “MERGE CUBES” <cube-alias-list> “INTO” <global-cube-alias>
[<set-operation>] “ON” <dim-attr-list>
([<preference-subclauses>] [<aggregation-subclauses>] | [<context-dimension-subclause>) <global-mapping-subclause>

<cube-alias-list> ::= <cube-alias> { [“,”]<cube-alias>}.

<dim-attr-list> ::= <local-dim-attr-id> { [“,”] <local-dim-attr-id>}

<preference-subclauses> ::= [“(“] <preference-subclause> { [“,”] <preference-subclause>} [“)”].

<preference-subclause> ::= “PREFER” < cube-alias> “.” <measure-attr-name>
(“DEFAULT” | “WHERE” <conditions>).

Kommentar [WB18]:
1 μ(g[N, prefer]h)

<context-dim-subclause> ::= [“(“] “TRACKING SOURCE AS DIMENSION” <dim-name> “(“ <schema-spec> “)”
(“IS” [“’”] <value> [“’”] “WHERE” <source-condition>
{“,” “IS” [“’”] <value> [“’”] “WHERE” <source-condition> } | “DEFAULT”) [“)”].

Kommentar [WB19]:
1 $\varepsilon_{A' = v}(c)$

<schema-spec> ::= <SQL DDL statement>.

<source-condition> ::= “SOURCE()” “=” [“’”] <value> [“’”].

<aggregation-subclauses> ::= [“(“] <aggregation-subclause> { [“,”] <aggregation-subclause>} [“)”].

<aggregation-subclause> ::= “AGGREGATE MEASURE” [<global-cube-alias>] “.” <measure-attr-name>
“IS” <aggregation-function> “OF” <local-measure-attr-id>
[“WHERE” <conditions>].

Kommentar [WB20]:
1 μ(g[N, (min|max|sum|avg)]h)

<global-mapping-subclause> ::= [“(“] <measure-mappings> [“,”] <dim-mappings> [“)”].

<measure-mappings> ::= <measure-mapping> { [“,”] <measure-mapping>}.

Kommentar [WB21]:
* $\lambda_{(N \subset MF)}$ (c)

<measure-mapping> ::= “MEASURE” <global-cube-alias> “.” <measure-attr-name>
[“->” <new-measure-attr-name>].

Kommentar [WB22]:
1 $\varsigma_{M' \leftarrow M}(c)$

<dim-mappings> ::= <dim-mapping> { [“,”]<dim-mapping>}.

Kommentar [WB23]:
1 $\pi_{(L \subset Ac)}$ (c)

<dim-mapping> ::= “DIM” <global-cube-alias> “.” <dim-attr-name> [“->” <new-dim-attr-name>].

Kommentar [WB24]:
1 $\varsigma_{A' \leftarrow A}(c)$

Anhang B

Integrationstest

Um die Verarbeitbarkeit der vom SQL-MDi Query Parser erzeugten Operator-Baumstruktur durch eine Prozessor-Komponente zu evaluieren, wurde ein Integrationstest implementiert, mit welchem die Funktionalität der Prozessor-Komponente simuliert und ein integriertes Ergebnis erzeugt werden kann (vgl. Abschnitt 8.1).

Der folgende Abschnitt dokumentiert jene SQL-MDi-Abfrage, deren entsprechende Operator-Baumstruktur evaluiert wurde (Abschnitt B.2 zeigt die Ausgabe der Baumstruktur des SQL-MDi Query Parser). Für Informationen über die Zuordnung von SQL-MDi-Anweisungen zu DA/FA-Operatoren wird auf Abschnitt 5.4 und Anhang A verweisen. Abschnitt B.2 präsentiert die zu integrierenden Komponentenschemata und -instanzen, nämlich „local data warehouse 1“ und „local data warehouse 2“, sowie das integrierte Ergebnis, d.h. den globalen, instanziierten Würfel („global data warehouse“). Die zwischen den lokalen DWS bestehenden Heterogenitäten können über die Nummerierung und die Bezeichnung des Konflikts identifiziert werden. Den SQL-MDi-Anweisungen des Listing B.1 und den Operatoren der Baumstruktur des Abschnitts B.2 ist die Nummer des aufzulösenden Konflikts als Kommentar zugeordnet.

B.1 SQL-MDi-Abfrage

```
1  DEFINE
2  CUBE mfuA::verkauf AS c1
3  (MEASURE c1.dauer, MEASURE c1.umsatz_telefonie, MEASURE c1.umsatz_sonstiges,
4  DIM c1.datum_std, DIM c1.svnr, DIM c1.mobilnetz,
5  PIVOT MEASURES c1.umsatz_telefonie, c1.umsatz_sonstiges
6    INTO c1.umsatz USING c1.umsatzkategorie)   -- 10.
7  (ROLLUP c1.datum_std TO LEVEL c1.datum[datum])  -- 9.
8
9  CUBE mfuB::verkauf AS c2
10 (MEASURE c2.gespraechsdauer -> dauer /* 4. */, MEASURE c2.umsatz,
11 DIM c2.datum
12   (MAP LEVELS c2.datum([datum],[month -> monat /* 5. */],[jahr])),  -- 2.
13 DIM c2.kunde -> svnr  -- 3.
14   (MAP LEVELS c2.kunde([kunde -> svnr /* 3. */],[tarif]),
15 DIM c2.mobilnetz, DIM c2.umsatzkategorie) -- 1.
16 (CONVERT MEASURES APPLY usd2Eur() FOR c2.umsatz DEFAULT)   -- 7.
17
18 GLOBAL CUBE dw0::verkauf AS c0
19
20 MERGE DIMENSIONS c1.datum AS d1, c2.datum AS d2 INTO c0.datum AS d0
21
22 MERGE DIMENSIONS c1.kunde AS d3, c2.kunde AS d4 INTO c0.kunde AS d5
23 (RELATE d3.svnr, d4.kunde WHERE d3.svnr=d4.kunde USING HIERARCHY OF d3) -- 12.
24 (MATCH ATTRIBUTES d3.bezeichnung IS d4.name)  -- 6.
25 (CONVERT ATTRIBUTES APPLY usd2Eur() FOR d4.grundgebuehr DEFAULT)  -- 8.
26
27 MERGE DIMENSIONS c1.mobilnetz AS d6, c2.mobilnetz AS d7 INTO c0.mobilnetz AS d8
28 (RENAME d6.mobilnetz >> 'HandyTelInc' WHERE c1.mobilnetz='HandyTel')  -- 11.
29
30 MERGE DIMENSIONS c1.umsatzkategorie AS d9, c2.umsatzkategorie AS d10
31   INTO c0.umsatzkategorie AS d11
32
33 MERGE CUBES c1,c2 INTO c0 ON datum, svnr, mobilnetz, umsatzkategorie
34 AGGREGATE MEASURE dauer IS SUM OF dauer,    -- 13.
35 AGGREGATE MEASURE umsatz IS SUM OF umsatz   -- 13.
36 (MEASURE dauer, MEASURE umsatz, DIM datum, DIM svnr, DIM mobilnetz, DIM umsatzkategorie)
```

Listing B.1: Integrationstest: SQL-MDi-Abfrage

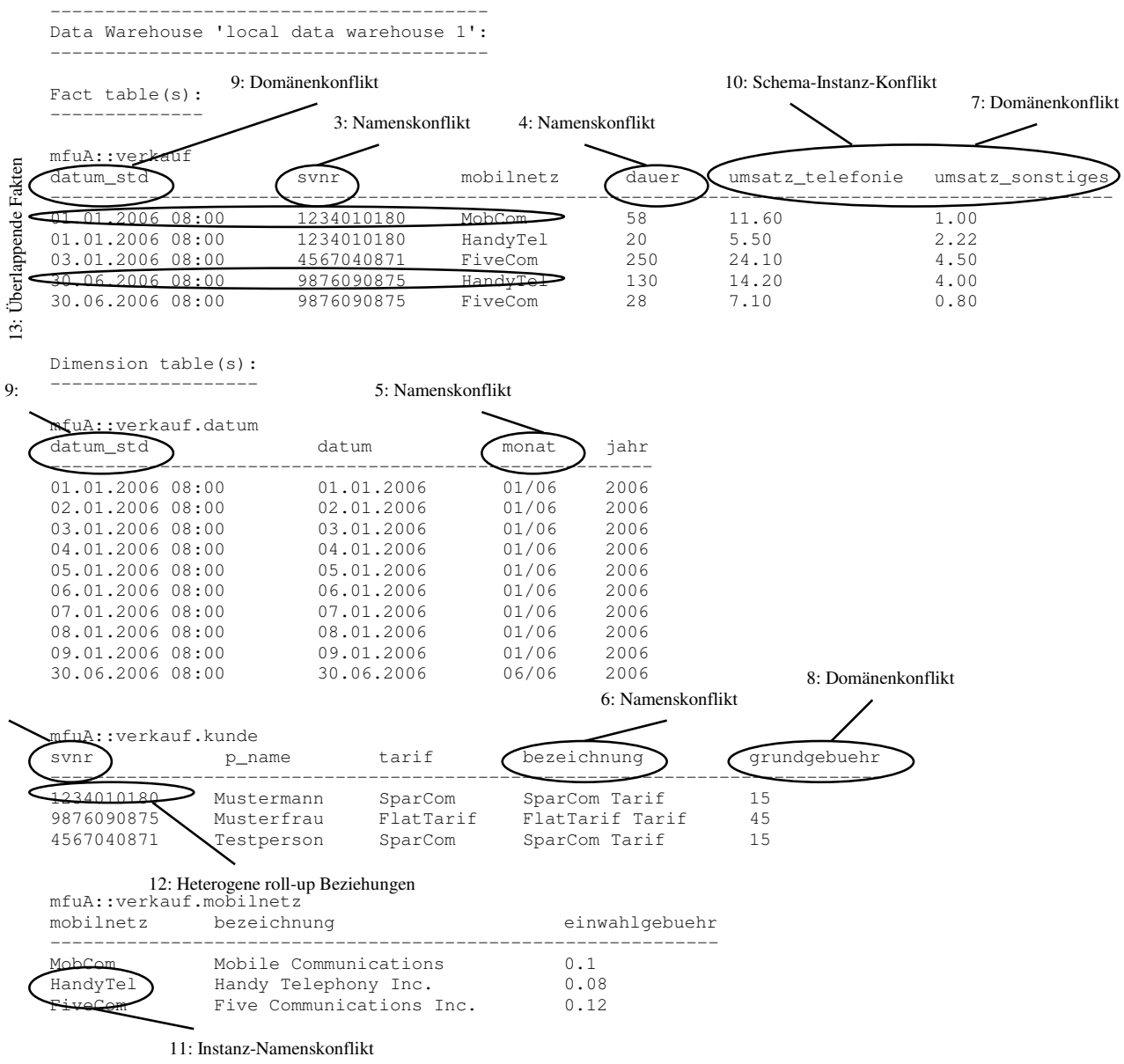
B.2 Operator-Baumstruktur

```
root
:
|−−DA MergeDim
:      |−− set-: null
:      |−− Dimension
:      :      |−− dimension: mfuA::verkauf.datum
:      :      |−− DA Delete level −− 9.
:      :      :      |−− level to delete: mfuA::verkauf.datum[datum_std]
:      |−− Dimension
:      :      |−− dimension: mfuB::verkauf.datum
:      :      |−− DA Delete level −− 2.
:      :      :      |−− level to delete: mfuB::verkauf.datum[quartal]
:      :      |−− DA Rename −− 5.
:      :      :      |−− deprecated attribute: mfuB::verkauf.datum[month]
:      :      :      |−− prefered attribute: mfuB::verkauf.datum[monat]
:      |−− Global Dimension
:      :      |−− global-dimension: dw0::verkauf.datum
:
|−−DA MergeDim
:      |−− set-operation: null
:      |−− Dimension
:      :      |−− dimension: mfuA::verkauf.kunde
:      |−− Dimension
:      :      |−− dimension: mfuB::verkauf.kunde
:      :      |−− DA Override rollup −− 12.
:      :      :      |−− deprecated level: mfuB::verkauf.kunde[kunde]
:      :      :      |−− prefered level: mfuA::verkauf.kunde[svnr]
:      :      :      |−− join conditions: mfuB::verkauf.kunde.kunde = mfuA::verkauf.kunde.svnr
:      :      |−− DA Convert −− 8.
:      :      :      |−− attrToConvert: mfuB::verkauf.kunde.grundgebuehr
:      :      :      |−− conversionFunction: usd2Eur()
:      :      |−− DA Rename −− 3.
:      :      :      |−− deprecated attribute: mfuB::verkauf.kunde[kunde]
:      :      :      |−− prefered attribute: mfuB::verkauf.kunde[svnr]
:      :      |−− DA Rename −− 6.
:      :      :      |−− deprecated attribute: mfuB::verkauf.kunde.name
:      :      :      |−− prefered attribute: mfuB::verkauf.kunde.bezeichnung
:      :      |−− Global Dimension
:      :      :      |−− global-dimension: dw0::verkauf.kunde
:
```

```
|– –DA MergeDim
:     |– – set-operation: null
:     |– – Dimension
:     :     |– – dimension: mfuA::verkauf.mobilnetz
:     :     |– – DA Change –– 11.
:     :     :     |– – deprecated attribute: mfuA::verkauf.mobilnetz.mobilnetz
:     :     :     |– – value: HandyTelInc
:     :     :     |– – predicate: mfuA::verkauf.mobilnetz=HandyTel
:     |– – Dimension
:     :     |– – dimension: mfuB::verkauf.mobilnetz
:     |– – Global Dimension
:     :     |– – global-dimension: dw0::verkauf.mobilnetz
:
|– –DA MergeDim
:     |– – set-operation: null
:     |– – Dimension
:     :     |– – dimension: mfuA::verkauf.umsatzkategorie
:     |– – Dimension
:     :     |– – dimension: mfuB::verkauf.umsatzkategorie
:     |– – Global Dimension
:     :     |– – global-dimension: dw0::verkauf.umsatzkategorie
:
|– –FA Merge facts
:     |– – set-operation: null
:     |– – merging-dimensions: dw0::verkauf.datum, dw0::verkauf.svnr,
:     |– – dw0::verkauf.mobilnetz, dw0::verkauf.umsatzkategorie
:     |– – Cube
:     :     |– – cube: mfuA::verkauf
:     :     |– – FA Project
:     :     :     |– – dimensional attributes: mfuA::verkauf.datum,
:     :     :     |– – mfuA::verkauf.svnr, mfuA::verkauf.mobilnetz
:     :     |– – FA Pivot (Merge measure attributes) –– 10.
:     :     :     |– – source-measures:
:     :     :     |– – mfuA::verkauf.umsatz_telefonie,
:     :     :     |– – mfuA::verkauf.umsatz_sonstiges
:     :     :     |– – new-measure: mfuA::verkauf.umsatz
:     :     :     |– – context-dim: mfuA::verkauf.umsatzkategorie
```

```
:   |-- Cube
:   :   |-- cube: mfuB::verkauf
:   :   |-- FA Project -- 1.
:   :   :   |-- dimensional attributes: mfuB::verkauf.datum, mfuB::verkauf.kunde,
:   :   :   |-- mfuB::verkauf.mobilnetz, mfuB::verkauf.umsatzkategorie
:   :   |-- FA Convert -- 7.
:   :   :   |-- measure to convert: mfuB::verkauf.umsatz
:   :   :   |-- conversion-function: usd2Eur()
:   :   |-- FA Rename -- 4.
:   :   :   |-- deprecated measure: mfuB::verkauf.gespraechsdauer
:   :   :   |-- prefered measure: mfuB::verkauf.dauer
:   :   |-- FA Rename -- 3.
:   :   :   |-- deprecatedAttr: mfuB::verkauf.kunde
:   :   :   |-- preferedAttr: mfuB::verkauf.svnr
:   |-- Global Cube
:   :   |-- global-cube: dw0::verkauf
:   :   |-- FA Merge facts (Aggregation) -- 13.
:   :   :   |-- aggregated measure: dw0::verkauf.dauer
:   :   :   |-- measures to combine: mfuA::verkauf.dauer, mfuB::verkauf.dauer
:   :   :   |-- aggregation-function: SUM
:   :   :   |-- group-by: dw0::verkauf.datum, dw0::verkauf.svnr,
:   :   :   |-- dw0::verkauf.mobilnetz, dw0::verkauf.umsatzkategorie
:   :   |-- FA Merge facts (Aggregation) -- 13.
:   :   :   |-- aggregated measure: dw0::verkauf.umsatz
:   :   :   |-- measures to combine: mfuA::verkauf.umsatz, mfuB::verkauf.umsatz
:   :   :   |-- aggregation-function: SUM
:   :   :   |-- group-by: dw0::verkauf.datum, dw0::verkauf.svnr,
:   :   :   |-- dw0::verkauf.mobilnetz, dw0::verkauf.umsatzkategorie
:   :   |-- FA Project
:   :   :   |-- dimensional attributes: dw0::verkauf.datum, dw0::verkauf.svnr,
:   :   :   |-- dw0::verkauf.mobilnetz, dw0::verkauf.umsatzkategorie
```

B.3 Testdaten und Testergebnis



```
---------------------------------------
Data Warehouse 'local data warehouse 2':
---------------------------------------

Fact table(s):
--------------

mfuB::verkauf
datum       kunde       mobilnetz   werbeaktion   umsatzkategorie   gespraechsdauer   umsatz
-------------------------------------------------------------------------------------------
01.01.2006  1234010180  MobCom      null          umsatz_telefonie  17                2.55
01.01.2006  1234010180  MobCom      Winter2006    umsatz_telefonie  23                8.50
01.01.2006  1234010180  MobCom      Winter2005    umsatz_telefonie  9                 7.50
30.06.2006  9876090875  HandyTelInc null          umsatz_telefonie  15                1.20
25.08.2006  6785041070  HandyTelInc Sommer2006    umsatz_telefonie  50                6.20
25.08.2006  6785041070  HandyTelInc Sommer2006    umsatz_sonstiges  0                 3.50


Dimension table(s):
-------------------

mfuB::verkauf.datum
datum       month   quartal   jahr
-----------------------------------
01.01.2006  01/06   1         2006
02.01.2006  01/06   1         2006
03.01.2006  01/06   1         2006
04.01.2006  01/06   1         2006
25.08.2006  08/06   3         2006

mfuB::verkauf.kunde
kunde       p_name      altersgruppe  tarif   name          grundgebuehr
-----------------------------------------------------------------------
1234010180  mustermann  31-40         FullCom FullCom Tarif 15
6785041070  testperson  21-30         2For0   2For0 Tarif   20

mfuB::verkauf.mobilnetz
mobilnetz     bezeichnung
-----------------------------------
MobCom        Mobile Communications
HandyTelInc   Handy Telephony Inc.
MobileCall    Mobile Calling Inc.
WirelessCom   Wireless Communications

mfuB::verkauf.werbeaktion
werbeaktion          werbetyp
-----------------------------
Weihnachten2006      TV
Sommer2006           Inserate
Winter2006           Inserate
Winter2005           Radio
```

1: Dimensionalität
2: Heterogene Aggregationshierarchien
3:
4:
5:
6:
7:
8:
9:
10:
11:
12:
13:

```
---------------------------------------
Data Warehouse 'global data warehouse':
---------------------------------------

Fact table(s):
--------------

dw0::verkauf
datum          svnr           mobilnetz        umsatzkategorie        dauer     umsatz
-------------------------------------------------------------------------------------------------------
01.01.2006     1234010180     MobCom           umsatz_telefonie       107.0     25.869230769230768
01.01.2006     1234010180     MobCom           umsatz_sonstiges       58        1.00
01.01.2006     1234010180     HandyTelInc      umsatz_telefonie       20        5.50
01.01.2006     1234010180     HandyTelInc      umsatz_sonstiges       20        2.22
03.01.2006     4567040871     FiveCom          umsatz_telefonie       250       24.10
03.01.2006     4567040871     FiveCom          umsatz_sonstiges       250       4.50
30.06.2006     9876090875     HandyTelInc      umsatz_telefonie       145.0     15.123076923076923
30.06.2006     9876090875     HandyTelInc      umsatz_sonstiges       130       4.00
30.06.2006     9876090875     FiveCom          umsatz_telefonie       28        7.10
30.06.2006     9876090875     FiveCom          umsatz_sonstiges       28        0.80
25.08.2006     6785041070     HandyTelInc      umsatz_telefonie       50        4.769230769230769
25.08.2006     6785041070     HandyTelInc      umsatz_sonstiges       0         2.692307692307692


Dimension table(s):
-------------------

dw0::verkauf.datum
datum          monat     jahr
-----------------------------
01.01.2006     01/06     2006
02.01.2006     01/06     2006
03.01.2006     01/06     2006
04.01.2006     01/06     2006
05.01.2006     01/06     2006
06.01.2006     01/06     2006
07.01.2006     01/06     2006
08.01.2006     01/06     2006
09.01.2006     01/06     2006
30.06.2006     06/06     2006
25.08.2006     08/06     2006


dw0::verkauf.kunde
svnr           p_name         tarif          bezeichnung          grundgebuehr
-----------------------------------------------------------------------------
1234010180     Mustermann     SparCom        SparCom              15
9876090875     Musterfrau     FlatTarif      FlatTarif Tarif      45
4567040871     Testperson     SparCom        SparCom Tarif        15
6785041070     testperson     2For0          2For0 Tarif          15.384615384615383


dw0::verkauf.mobilnetz
mobilnetz        bezeichnung
-----------------------------------
MobCom           Mobile Communications
HandyTelInc      Handy Telephony Inc.
FiveCom          Five Communications Inc.
MobileCall       Mobile Calling Inc.
WirelessCom      Wireless Communications
```