Gunter Saake
Kai-Uwe Sattler
Andreas Heuer

Datenbanken

Konzepte und Sprachen

Sechste Auflage

Dieses pdf-Kapitel ist eine kostenlose Ergänzung zum oben genannten Buch, das 2018 bei MITP erschienen ist.

Bitte beachten: Verweise auf Umgebungen B-X beziehen sich auf Teile innerhalb dieses Download-Kapitels. Verweise, die mit Kapitelnummern beginnen, wie 11-31, beziehen sich auf Umgebungen innerhalb des obigen Lehrbuches. Verweise auf Seiten vor 738 beziehen sich ebenfalls auf das zugrundeliegende Lehrbuch.

Literaturhinweise in diesen pdf-Kapiteln beziehen sich auf ein erweitertes Literaturverzeichnis zum Lehrbuch, das auch als kostenlose Ergänzung an derselben Stelle zum Download bereitsteht.

mitp

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

Bibliografische Information der Deutschen Nationalbibliothek
Die Deutsche Nationalbibliothek verzeichnet diese Publikation in der Deutschen Nationalbibliografie; detaillierte bibliografische Daten sind im Internet über <http://dnb.d-nb.de> abrufbar.

Bei der Herstellung des Werkes haben wir uns zukunftsbewusst für umweltverträgliche und wiederverwertbare Materialen entschieden. Der Inhalt ist auf elementar chlorfreiem Papier gedruckt.

ISBN 978-3-95845-776-8
6. Auflage 2018

www.mitp.de
E-Mail: mitp-verlag@sigloch.de

Telefon: +49 7953/7189-079
Telefax: +49 7953/7189-082

© 2018 mitp Verlags GmbH & Co. KG, Frechen

Dieses Werk, einschließlich aller seiner Teile, ist urheberrechtlich geschützt. Jede Verwertung außerhalb der engen Grenzen des Urheberrechtsgesetzes ist ohne Zustimmung des Verlages unzulässig und strafbar. Dies gilt insbesondere für Vervielfältigungen, Übersetzungen, Mikroverfilmungen und die Einspeicherung und Verarbeitung in elektronischen Systemen.

Die Wiedergabe von Gebrauchsnamen, Handelsnamen, Warenbezeichnungen usw. in diesem Werk berechtigt auch ohne besondere Kennzeichnung nicht zu der Annahme, dass solche Namen im Sinne der Warenzeichen- und Markenschutz-Gesetzgebung als frei zu betrachten wären und daher von jedermann benutzt werden dürften.

Lektorat: Sabine Janatschek, Ernst-Heinrich Pröfener
Sprachkorrektorat: Jürgen Dubau, Astrid Langen
Covergestaltung: Christian Kalkert, www.kalkert.de
Bildnachweis Cover: iStock.com/Shawn Hempel | fotolia.com/karpenko_ilia
Satz: Gunter Saake, Magdeburg; Kai-Uwe Sattler, Ilmenau; Andreas Heuer, Rostock
Druck: Westermann Druck Zwickau GmbH

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

B.4 Objektorientierte und objektrelationale Modelle inklusive SQL:2003

Noch vor wenigen Jahren war ein großer Teil jedes Datenbanklehrbuchs dem Thema *objektorientierte Datenbanksysteme* (kurz ODBMS) gewidmet. Nun ist es still um die reinen ODBMS geworden, auch wenn einige Produkte weiterhin existieren. Viele der guten Neuerungen der ODBMS-Welle sind inzwischen unter dem Stichwort *objektrelational* in der SQL-Welt auch vorhanden. Wir werden zunächst die reinen ODBMS und den zugehörigen Standard der ODMG beschreiben und uns danach intensiv mit den objektrelationalen Konzepten auseinandersetzen, die in die Standards SQL:1999 und SQL:2003 eingeflossen sind.

B.4.1 Exkurs: Objektorientierte Datenbankmodelle

Ende der 80er Jahre des letzten Jahrhunderts kamen die ersten objektorientierten Datenbanksysteme auf den Markt. Ein Objektdatenbankmanagementsystem (kurz ODBMS) ist ein DBMS, das ein Objektdatenbankmodell unterstützt. Ein Objektdatenbankmodell wiederum ist ein Objektmodell (angelehnt an objektorientierte Programmiersprachen) mit datenbankspezifischen Eigenschaften wie Persistenz und Transaktionen.

Nach der Begeisterung über die neuen Systeme kam die bei Informatik-Innovationen verbreitete Ernüchterung, als sich zeigte, dass die neuen Vorteile nicht umsonst zu haben waren und einige bekannte DBMS-Vorteile nur schwer mit der Objektorientierung zu vereinbaren waren. Aktuell spielen ODBMS im Datenbankmarkt nur eine bescheidene Rolle. Die Zukunft wird zeigen, ob die Konzepte reiner ODBMS so tragfähig sind, dass eine Renaissance der Objektdatenbanken entsteht.

B.4.1.1 Objektorientierte Datenbanken

Wie bereits erwähnt entstanden die ersten ODBMS in den 80er Jahren des letzten Jahrhunderts. Motivation war damals der sogenannte *Impedance Mismatch*: Gerade technische Anwendungen wurden vermehrt in objektorientierten Programmiersprachen wie Smalltalk, CLOS, C++ und später Java programmiert. Objekte in derartigen Sprachen hatten Attribute, genau wie klassische Datenbanktupel, besaßen aber auch eine systemkontrollierte Identität, waren in Vererbungshierarchien eingeordnet und zu komplexen Objekten zusammengesetzt. All dies stand in Konflikt zu den relationalen Datenbanksystemen: Tupel sind dort durch nutzervergebene Schlüssel identifiziert, waren flach (erste Normalform), und Vererbungshierarchien gab es nur in Entwurfsdatenmodellen.

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

Die ersten ODBMS waren stark an Programmiersprachen, insbesondere C++ oder Smalltalk, angelehnt, und erweiterten diese durch eingeschränkte Datenbankfunktionaliät. Ein sie vereinendes einfaches Datenbankmodell wie zwanzig Jahre davor das relationale Datenmodell existierte nicht. Dementsprechend heterogen waren die Systeme. Nach diesen ersten Systemen kam eine Phase der Vereinheitlichung: Verschiedene Gruppen veröffentlichten Anforderungslisten an ODBMS, und der ODMG-Standard zur Vereinheitlichung der Syntax entstand.

ODBMS-Manifesto

Die bekannteste Anforderungsliste dieser Zeit war zweifelsohne das ODBMS-Manifesto [ABD^{+}89]. Es legte eine Reihe von Konzepten fest, die ein objektorientiertes Datenbanksystem und die mit ihm verbundenen Sprachen auf jeden Fall unterstützen mussten. Diese sogenannten *Golden Rules* waren die folgenden:

- Komplexe Objekte
- Objektidentität
- Kapselung
- Typen und Klassen
- Klassen- oder Typhierarchie
- Overriding, Overloading, Late Binding
- Berechnungsvollständige DB-Programmiersprache

Zu den Golden Rules gehörten außerdem die folgenden Punkte, die für DBMS eigentlich selbstverständlich sind: Erweiterbarkeit, Persistenz, Sekundärspeicherverwaltung, Synchronisation und Recovery von Transaktionen sowie Anfragesprachen.

Neben den Golden Rules nennt das ODBMS-Manifesto noch die sogenannten Goodies, also Funktionaliät, die wünschenswert ist, aber nicht in allen ODBMS verwirklicht sein müsste: Mehrfachvererbung, statische Typisierung und Typinferenz als Eigenschaften objektorientierter Programmiersprachen und Verteilung, Entwurfstransaktionen und Versionen als DBMS-Funktionalitäten.

Interessanterweise gibt es eine Reihe unberücksichtigter DBMS-Eigenschaften in diesem Manifesto, so Integrität, Sichten, Schemaevolution und Zugriffskontrolle.

Die genannten Vereinheitlichungsbestrebungen mündeten schließlich in dem ODMG-Standard, den wir im folgenden Abschnitt behandeln werden.

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

B.4.1.2 Der ODMG-Standard

Der ODMG-Standard geht auf die Gründung der Object Database Management Group (ODMG) als Untergruppe der OMG zurück. Mitglieder waren primär die damaligen Hersteller von ODBMS, es handelte sich um kein offizielles Standardisierungskomitee, sondern um einen Industriestandard. Die beteiligten Hersteller deckten einen Großteil des ODBMS-Marktes ab. 1993 erschien der erste ODMG-Industriestandard, das Jahr 2000 sah mit ODMG 3.0 [CB00] den bisher letzten Stand.

Der Standard bildet ein Art Kompromiss zwischen den damals existierenden ODBMS, OOPL wie C++ und Smalltalk sowie Konzepten von Forschungsprototypen. Festgelegt wurden

- ein Objektmodell inkl. DB-Konzepte (Persistenz, Transaktions- und Datenbankverwaltung),
- die Objektdefinitionssprache ODL (Erweiterung der OMG-IDL),
- die Objektanfragesprache OQL als Anpassung/Erweiterung von SQL,
- die Programmiersprachenanbindung (Binding): C++, Smalltalk, Java und
- ein Objektaustauschformat OIF.

Im Objektmodell wird zwischen Typen und Klassen unterschieden. Eine Typspezifikationen (ODL) legt mit der Schnittstellendefinition ausschließlich das abstrakte Verhalten fest. Klassendefinitionen hingegen spezifizieren sowohl das abstrakte Verhalten als auch einen abstrakten Zustand. Mit `interface` werden dabei abstrakte, nicht instantiierbare Typen spezifiziert, während mit `class` Klassen deklariert werden, von denen Instanzen erzeugt werden können.

Bei der Definition von Klassen wird in der ODL die Schnittstellenbeschreibung (Attribute, Beziehungen, Methodensignaturen) festgelegt. Die Implementierung von Klassen hingegen erfolgt ausschließlich in der Programmiersprachenanbindung. Das optionale Schlüsselwort `extent` legt eine persistente Extension fest, also einen Container für in der Datenbank zu speichernde Instanzen. Bei Instantiierung wird dann ein Objekt automatisch in die Extension eingetragen. Die Extensionen sind wichtig für datenbanktypische Funktionen wie Anfragen, Schlüsselbedingungen oder Indexverwaltung zur Unterstützung einer effizienten Suche.

◀Beispiel B-7▶ Das folgende Beispiel zeigt eine einfache Klassendeklaration in der ODL:

```
class Wein (
     extent WeinExtension) {
  attribute long weinID;
  attribute string Name;
```


© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

```
  attribute int Jahrgang;
  enum { Rot, Weiß, Rose } Farbe;
  attribute date Geburtstag;
  attribute list⟨string⟩ Rebsorte;
  ...
};
```

□

Anhand dieses Beispiels kann gezeigt werden, wozu Extensionen genutzt werden können. Die Extension ermöglicht die Angabe von Schlüsselbedingungen wie im folgenden Beispiel.

◀Beispiel B-8▶ Wir erweitern hierzu unser obiges Beispiel um die Schlüsselbedingung:

```
class Wein (
     extent WeinExtension,
     keys weinID, (Name, Jahrgang)) {
  attribute long weinID;
  attribute string Name;
  attribute int Jahrgang;
  enum { Rot, Weiß, Rose } Farbe;
  attribute date Geburtstag;
  attribute list⟨string⟩ Rebsorte;
  ...
};
```

□

Das Extensionskonzept ist notwendig, da das Klassenkonzept an Programmiersprachen angelehnt ist, die üblicherweise keine globalen Prüfungen auf Extensionen ermöglichen.

In der objektorientierten Welt sind die sogenannten *Klassenbeziehungen* das Gegenstück zu den Relationship-Typen des ER-Modells. Im Gegensatz zu UML beschreibt der ODMG-Standard ein Implementierungsmodell angelehnt an objektorientierte Programmiersprachen. Beziehungen zwischen Objekten werden daher über *Referenzattribute* realisiert. Das Schlüsselwort **relationship** deklariert derartige Beziehungen. Möglich sind nur binäre Beziehungen ohne Beziehungsattribute. ODMG kennt im Gegensatz zur objektorientierten Modellierung keine Aggregationsbeziehungen – existentielle Abhängigkeiten und Propagierung von Löschoperationen müssen daher explizit operational definiert werden.

ODMG kennt uni- und bidirektionale Beziehungen. Das Schlüsselwort **inverse** definiert die Rückrichtung einer bidirektionalen Beziehung. Referentielle Integrität wird dabei garantiert. Kardinalitäten werden nicht explizit spe-

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

zifiziert, können aber über die bei mengen- oder listenwertigen Beziehungen eingesetzten Kollektionstypen realisiert werden.

◀**Beispiel B-9**▶ Der folgende ODL-Ausschnitt zeigt die Deklaration einer inversen Beziehung. Durch die Wahl des Datentypkonstruktors **list** wird eine Ordnung auf der mehrwertigen Seite festgelegt.

```
class Wein {
   ...
   relationship Erzeuger produziertVon
        inverse Erzeuger::produziert;
};

class Erzeuger {
   ...
   relationship list⟨Wein⟩ produziert
        inverse Wein::produziertVon;
};
```

□

Ein objektorientiertes Modell muss auch die Spezialisierung unterstützen. ODMG spezifiziert sowohl den intensionalen als auch den extensionalen Aspekt (Letzteres über Extents) in einer kombinierten Spezialisierungshierarchie. Auf der intensionalen Seite wird eine Spezialisierung von Subtypen über Tupelerweiterung bzw. Redefinition von Operationen (erfordert dynamisches Binden) realisiert.

Für Schnittstellen kann wie gewohnt eine Typhierarchie aufgebaut werden, mit der die ISA-Beziehung nachgebildet werden kann. Mehrfachvererbung ist möglich.

```
interface Erzeuger { ... };
interface Händler : Erzeuger { ... };
```

Klassen können mehrere Schnittstellen haben. Aufgrund des extensionalen Aspekts ist nur eine Einfachvererbung zwischen Klassen erlaubt. Die **extends**-Beziehung realisiert dies.

```
class Erzeuger { ... };
class PrivatErzeuger extends Erzeuger { ... };
```

Da in ODBMS Objekte der Programmiersprache in die Datenbank bewegt werden sollen, ist die Frage der *Persistenz* von Objekten zu klären. In einer Anwendung gibt es sowohl *transiente* als auch *persistente* Objekte. Persistente Objekte überleben eine Anwendungssitzung in der Datenbank, transiente Objekte sind reine Laufzeitobjekte im Hauptspeicher.

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

Persistenzfähige Klassen können dabei prinzipiell sowohl transiente als auch persistente Objekte enthalten. Das DBMS muss daher durch Sprachmittel informiert werden, welche Objekte tatsächlich persistent sind. Die konkrete Realisierung ist dabei abhängig von System und Sprachanbindung. Beispielsweise gibt es in C++ eine persistente Wurzelklasse `d_Object` (typabhängige Persistenz), während in Java eine Verarbeitung durch einen Prozessor erfolgt (typorthogonale Persistenz).

B.4.1.3 OQL

Teil des ODMG-Standards ist die Anfragesprache OQL. OQL ist als objektorientierte, deklarative Anfragesprache konzipiert und basiert auf der Anfragesprache O_2SQL des O_2-Systems. OQL bietet eine Teilmenge der SQL-Konstrukte plus einer Reihe von ODBMS-spezifischen Erweiterungen:

- OQL ermöglicht die Nutzung von Objektidentitäten, Pfadausdrücken, Methoden und komplex strukturierten Werten.
- Anfragen können auf Mengen (z.B. Extensionen) und beliebigen Kollektionen ausgeführt werden.

Die Bedeutung einer Anfrage kann mittels relationaler, objekterhaltender und objektgenerierender Semantik festgelegt werden: Anfrageergebnisse können somit übliche SQL-Tabellen, Kollektionen aus der Datenbank extrahierter Objekte oder auch neu generierte Objekte sein.

Einen vollständige Beschreibung von OQL würde den Rahmen dieses Buches sprengen. Wir diskutieren daher nur einige ausgewählte Aspekte.

OQL nutzt den von SQL her bekannten SFW-Block. Im **`select`**-Teil ist der Aufruf von Methoden und Unterfragen und die Konstruktion komplexer Ergebnistypen möglich. Mittels **`distinct`** wird Mengensemantik erzwungen.

◀**Beispiel B-10**▶ Ein Beispiel ist die folgende Anfrage, die zu jedem Weingut die Multimenge der produzierten Weine liefert:

```
select distinct struct (e.Weingut, Weine:(
   select w.Name
   from e.produziert w))
from Erzeuger e
```

Der Ergebnistyp dieser Anfrage lautet:

```
set⟨struct⟨Weingut: string, Weine: bag⟨string⟩⟩⟩
```

□

Im **`from`**-Teil kann die Angabe einer Objekt- oder Wertekollektion als Klassenextension, ein mengenwertiges Referenzattribut, komplexes Attribut,

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

Ergebnis eines Methodenaufrufs oder Unterabfrage stehen. Um die SQL-Semantik zu erreichen, erfolgt eine automatische Umwandlung von `list` und `array` nach `bag`.

◀**Beispiel B-11**▶ Auch hier wieder ein kurzes Beispiel:

```
select w.Name
from (select e.produziert
   from Erzeuger e
   where e.Weingut = 'Château La Rose') w
```

□

Der `where`-Teil entspricht den SQL-Konventionen.

In objektorientierten Programmiersprachen spielen *Pfadausdrücke* eine besondere Rolle. Das Navigieren entlang Pfaden ersetzt den Verbund relationaler Anfragen. Eine OQL-Anfrage kann (beliebig lange) Pfadausdrücke beinhalten, etwa als Beispiel wie folgt (ausgehend von einem benannten Objekt `einWein`):

```
einWein.produziertVon.Weingut
```

Hierbei handelt es sich übrigens um eine vollständige Anfrage – eine OQL-Anfrage muss nicht unbedingt einen SFW-Block enthalten. In OQL gilt die Einschränkung, dass Zwischenelemente des Pfads einelementige Referenzen sein müssen. Daher wäre Folgendes nicht erlaubt:

```
einErzeuger.produziert.Jahrgang
```

Stattdessen würde man in OQL wie folgt formulieren:

```
select w.Jahrgang
from Erzeuger e, e.produziert w
```

Trotz interessanter Konzepte hat OQL kaum Verbreitung gefunden. Nur das O_2-System bot eine weitgehende Unterstützung von OQL, während andere ODBS nur rudimentäre Umsetzung enthielten oder sogar vollständig auf eine deklarative Anfragesprache verzichteten. Einige der mit OQL eingeführten Ideen finden aber eine Renaissance in den Anfragesprachen der objektrelationalen Mapper und JDO (siehe Abschnitt 14.4.3) und auch XQuery (siehe Abschnitt 20.5).

B.4.2 Abbildung von Objekten auf Relationen

Die Entwicklung objektrelationaler Datenmodelle verfolgt zwei Ziele:

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

- die Integration objektorientierter Konzepte in das erfolgreiche Relationenmodell sowie
- die Nutzung stabiler relationaler Datenbanktechnologie für objektorientierte Anwendungen.

Der letztere Punkt erfordert die Abbildung von Objekten und Klassen auf Relationen. Wir werden Aspekte dieser Abbildung in diesem Abschnitt diskutieren. Die tatsächliche Abbildung kann explizit (sogenanntes OR-Mapping, siehe auch Abschnitt 14.4) oder implizit durch ein DBMS intern erfolgen. In jedem Fall erfordert eine solche Abbildung die Berücksichtigung folgender Modellierungskonstrukte:

- die Standardkonstrukte wie Klasse oder Beziehung, die auch im ER-Modell vorhanden sind und deren Abbildung wir bereits in Abschnitt 14.4 behandelt haben,
- die Typkonstruktoren wie **`list`**, **`set`** oder **`array`**,
- Spezialisierungshierarchien.

Die folgenden Abschnitte behandeln nun die zwei letzteren Konzepte von Objektmodellen.

B.4.2.1 Typkonstruktoren

Typkonstruktoren werden im Detail noch später in diesem Kapitel behandelt, doch die folgenden Ausführungen erfordern noch keine tiefere Beschäftigung mit ihren Konzepten. Die wichtigsten Typkonstruktoren konstruieren Listen mittels **`list`**, Mengen mittels **`set`**, Multimengen mittels **`bag`** und Arrays mittels **`array`**. Zum Einsatz kommen sie sowohl bei „normalen“ Attributen wie auch bei Referenzattributen zur Realisierung von Beziehungen. Sie können geschachtelt auftreten.

Abbildung B.48 zeigt die Abbildung eines mengenwertigen Attributes. Hier wird eine separate Tabelle erzeugt, die die einzelnen Werte enthält. Fremdschlüsselbeziehungen garantieren die Zusammensetzbarkeit der Originaltabelle. Bei den verschiedenen Konstruktoren sind deren Eigenheiten zu beachten:

- Bei Listen muss ein weiteres Attribut die Listenposition aufnehmen, sofern die Reihenfolge von Bedeutung ist.
- Bei Multimengen gilt analoges für die Multiplizität des Wertes.
- Bei Arrays fester Länge kann man alternativ überlegen, bei einer Länge n das Attribut $n-1$-mal zu kopieren. So erhält man dann beispielsweise für den Fall in Abbildung B.48 Attribute `Winzer1` bis `Winzer4` (bei $n=4$).

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

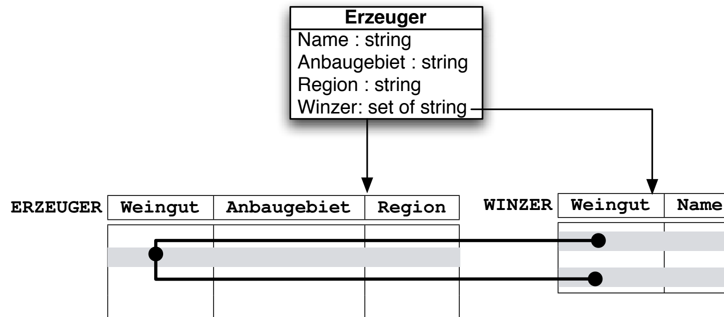

Abbildung B.48: Abbildung eines mengenwertigen Attributs

Eine ganz anders geartete Abbildung ist, die komplexen Attributwerte einfach binär oder als String serialisiert in ein Attribut abzulegen. Dies kann unter Umständen Performanzgewinne bedeuten, aber bereitet natürlich zusätzlichen Aufwand bei der Anfragebearbeitung, wenn dieses Attribut angefragt sein sollte.

B.4.2.2 Abbildung der Spezialisierungshierarchie

Bei der Abbildung der Spezialisierungshierarchie muss sowohl der extensionale als auch der intensionale Aspekt berücksichtigt werden. Es gibt drei Standardvarianten der Abbildung, von denen auch Mischformen auftreten können:

- Die horizontale Partitionierung wird insbesondere eingesetzt, wenn nur die Blätter einer Hierarchie Instanzen haben. Hierbei werden alle Attribute der Superklassen in die Tabellen der Blätter aufgenommen.
- Bei der vertikalen Partitionierung wird jeder Klasse der Hierarchie eine Tabelle zugeordnet, die jeweils nur die lokalen Attribute hat.
- Die typisierte Partitionierung fasst alle Klassen der Hierarchie in einer einzigen Tabelle mit einem Attribut als Typdiskriminator (und Nullwerten in den jeweils nichtzutreffenden Attributen bei Instanzen) zusammen.

Wir werden diese drei Varianten jeweils kurz anhand eines Beispiels erläutern.

Horizontale Partitionierung

Die horizontale Partitionierung erzeugt für jede Klasse eine Tabelle mit allen (originären und ererbten) Attributen. Horizontal bezieht sich darauf, dass Instanzen als Ganze verteilt werden (analog zum Verteilungsentwurf). Eine In-

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

stanz wird in derjenigen Tabelle gespeichert, die der speziellsten Klasse entspricht, der sie angehört. Faktisch wird dieser Ansatz dann eingesetzt, wenn nur Instanzen in den Blättern existieren, und daher keine Tabellen für innere Klassen der Hierarchie benötigt werden.

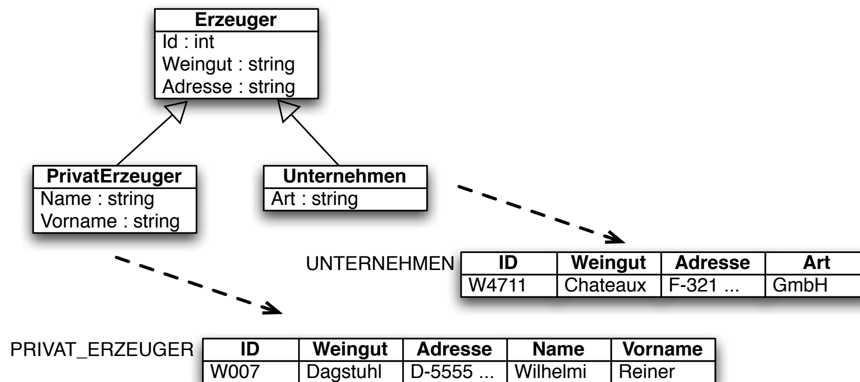

Abbildung B.49: Abbildung von Spezialisierungen: Horizontale Partitionierung

Abbildung B.49 zeigt ein Beispiel für die horizontale Partitionierung. Hier werden nur die Blätter der Vererbungshierarchie als Tabellen repräsentiert. Jede Tabelle enthält die Attribute der korrespondierenden Klasse und alle Attribute der Superklasse (bzw. Superklassen).

Dieser Ansatz ist vorteilhaft bei abstrakten Superklassen bzw. vollständigen Partitionierungen einer Klasse in Subklassen. Allerdings ist bei Anfragen über Superklassen der Zugriff auf mehrere Tabellen notwendig.

Vertikale Partitionierung

Die vertikale Partitionierung erzeugt für jede Klasse eine Tabelle, die nur die jeweils lokal definierten Attribute aufweist. Instanzen müssen nun auf mehrere Tabellen aufgeteilt werden.

Abbildung B.50 zeigt ein Beispiel. Jeder Klasse der Vererbungshierarchie wird eine eigene Tabelle wie folgt zugeordnet: die Tabellen der abgeleiteten Klassen umfassen die spezielle Attribute der korrespondierenden Klasse sowie den Primärschlüssel. Ein Objekt wird aus mehreren Tupeln entlang der Vererbungshierarchie zusammengesetzt.

Dieser Ansatz ermöglicht effiziente Anfragen auf Attributen der Superklassen bzw. auf speziellen Attributen der abgeleiteten Klassen. Der Zugriff auf vollständige Objekte hingegen ist ineffizient.

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

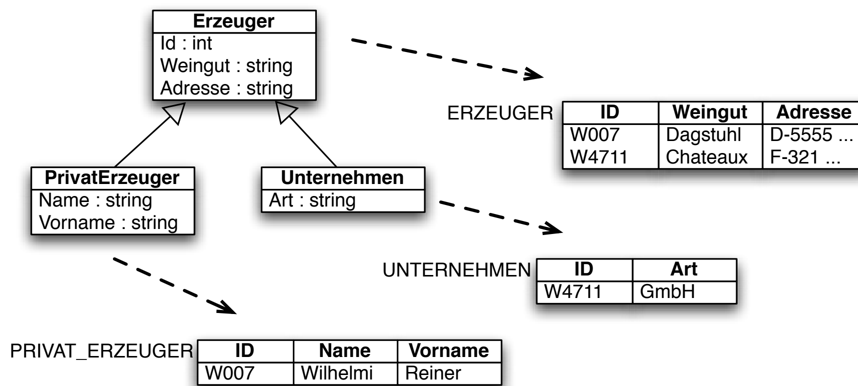

Abbildung B.50: Abbildung von Spezialisierungen: Vertikale Partitionierung

Typisierte Partitionierung

Die typisierte Partitionierung ist eigentlich gar keine Partitionierung: alle Instanzen der Hierarchie werden in einer einzigen Tabelle gespeichert. Die Zuordnung zur Klasse erfolgt durch ein Typisierungsattribut.

Abbildung B.51 zeigt eine typisierte Partitionierung. Dieser Ansatz erlaubt effiziente Anfragen, speichert aber ggf. viele Nullwerte ab.

Tabelle B.6 fasst die Eigenschaften der drei Partitionierungsarten im Vergleich zusammen.

Partitionierung	**Anfragen auf Superklassen**	**Anfragen auf Subklassen**
horizontal	ineffizient	effizient
vertikal	effizient	ineffizient
typisiert	effizient, aber unflexibel	effizient, aber unflexibel

Tabelle B.6: OR-Mapping von Hierarchien: Vergleich der Partitionierungsarten

B.4.3 Objektrelationale Erweiterungen

Das objektrelationale Datenmodell ist nicht so einheitlich entstanden wie das relationale. Verschiedene Hersteller von RDBMS haben unterschiedliche Erweiterungen eingebaut, die Konzepten von ODBMS entsprechen. Der aktuelle

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

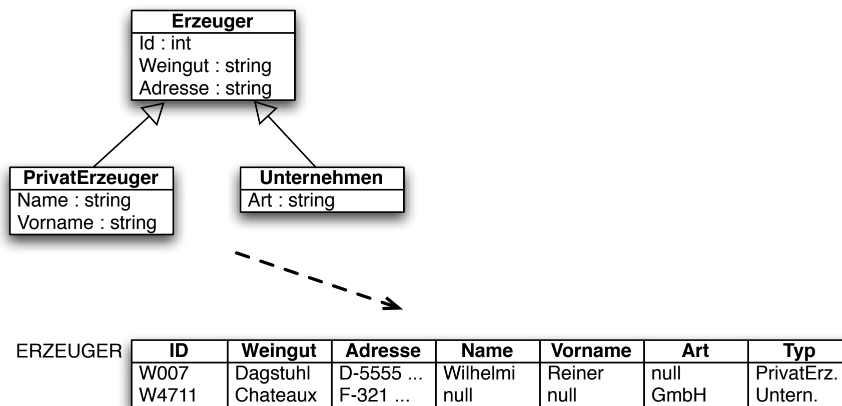

Abbildung B.51: Abbildung von Spezialisierungen: Typisierte Partitionierung

SQL-Standard kann als Referenz dienen, lässt aber noch einige Punkte offen (etwa im Bereich Kollektionsdatentypen). Wir werden in diesem Abschnitt die Konzepte vorstellen, die allgemein unter „objektrelational" subsumiert werden, und uns dann anschließend dem SQL-Standard als konkrete Ausprägung zuwenden.

B.4.3.1 Large Objects: BLOB und CLOB

Im Rahmen der Entwicklung von objektrelationalen Konzepten wurde das bisher eher eingeschränkte Typkonzept von SQL erweitert. Neben nutzerdefinierten Datentypen, Datentypen für Objektreferenzen und Datentypkonstruktoren werden auch die sogenannten Large Objects zu den Erweiterungen gezählt, auch wenn diese Konzepte trotz ihres Namens nicht direkt aus objektorientierten Modellen (und nur diesen) abgeleitet sind.

Large Objects (kurz LOB) treten als Binary LOB, kurz BLOB, und Character LOB, kurz CLOB auf. Ein LOB ist ein langer Bit- oder Character-String mit speziellen Operationen. Einsatzgebiete für BLOBs sind Daten im Binärformat wie Bilder oder Tondokumente, während CLOBs für die Dokumentverwaltung eingesetzt werden.

Bei LOB ist die Angabe der maximalen Größe erforderlich, wobei die Grenzen systemspezifisch sind. Im folgenden SQL-Code wird die `WEINE`-Tabelle um ein Foto-Attribut erweitert.

```
create table WEINE (
   ...
   Foto blob(100K))
```


© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

B.4.3.2 Typkonstruktoren

Eine wichtige objektrelationale Erweiterung ist der Einsatz von Typkonstruktoren. Sie werden auch als Kollektionsdatentypen bezeichnet, weil sie Kollektionen von gleichartigen Daten modellieren, die analog zu Tabellen als Ganzes angefragt und manipuliert werden können.

Parametrisierbare Datentypen

Aus dem Gebiet der abstrakten Datentypen kennt man die parametrisierbaren Datentypen. Im Datenbankbereich sind insbesondere die folgenden parametrisierbaren Datentypen von Interesse:

- der Tupel-Typkonstruktor,
- der Set-Typkonstruktor für Mengen,
- der Bag-Typkonstruktor für Multimengen,
- der List-Typkonstruktor,
- der Array-Typkonstruktor.

Diese parametrisierten Datentypen heißen auch *Typkonstruktoren* und erlauben eine Typkonstruktion auf Basis vorhandener Datentypen (den Eingangsdatentypen). Prinzipiell ist die Orthogonalität der Anwendung der Typkonstruktoren gegeben: Listen von Mengen sind so konstruierbar. Die Array-, List-, Bag- und Set-Typkonstruktoren werden auch *Kollektionsdatentypen* genannt, da sie eine Kollektion gleichartiger Elemente beschreiben, über die ein Iterator – oder ein SFW-Block – definiert werden kann. Wir werden nun diese Typkonstruktoren der Reihe nach vorstellen.

Tupel-Typkonstruktor

Der Tupelkonstruktor[3] ist aus Programmiersprachen bekannt. Er setzt einen Datenwert aus möglicherweise heterogenen Einzelwerten zusammen. Der Zugriff auf die Komponenten erfolgt über symbolische Namen. Ein so konstruiertes Tupel entspricht so einer Zeile einer relationalen Tabelle. Daher wird der Tupelkonstruktor statt als `tuple` auch als `row` bezeichnet.

Der *Tupel-Typkonstruktor* `tuple` kombiniert eine feste Anzahl typisierter Datenkomponenten zu einem komplexen Wert. Mathematisch entspricht dies der Idee des kartesischen Produktes. Die Identifizierung der einzelnen Komponenten erfolgt durch Feldnamen.

[3] Die korrekte Bezeichnung Tupel-Typkonstruktor wird in der Regel – analog auch bei den folgenden Konstruktoren – verkürzt zu Tupelkonstruktor, auch wenn der *Typ* konstruiert wird.

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

Die Operationen beschränken sich auf Operationen zum Erzeugen (ebenfalls notiert als **tuple**) von Tupeln und den aus Progarmmiersprachen bekannten Komponentenzugriffen mit der Punktnotation (Operator **.**). **tuple** entspricht dem **record** oder **struct** in Programmiersprachen. In den meisten Datenmodellen gibt es vordefinierte Tupel-Datentypen, so `Datum` und `Zeit`.

◀Beispiel B-12▶ Der Standarddatentyp `Datum` kann wie folgt als Tupel-Typ definiert werden:

```
Datum: tuple(Tag: integer,
             Monat: integer,
             Jahr: integer)
```

□

Set-Typkonstruktor

Der Mengenkonstruktor ist in Datenmodellen sicher der bekannteste Konstruktor, da er dem relationalen Datenmodell zugrundeliegt. Eine *Menge*, engl. *Set*, ist eine homogene Kollektion ohne Duplikate. Der *Set-Typkonstruktor* **set** spezifiziert endliche, homogene Mengen aus Werten eines Eingangsdatentyps. Er realisiert also die Zusammenfassung gleichartigen Elemente zur ungeordneten Menge (ohne Duplikate).

Mathematisch entspricht der Wertebereich eines so konstruierten Mengentyps der Potenzmenge über den Wertebereich des Eingangsdatentyps.

◀Beispiel B-13▶ Typisches Beispiel für einen Einsatz des **set**-Konstruktors ist die Deklaration eines Datentyps für ein Attribut, das eine Menge von Telefonnummern verwalten soll.

```
Telefone: set(string)
```

□

Auf den mit diesem Konstruktor konstruierten Mengen sind eine Reihe von Operationen definiert:

- Erzeugen (**set**) einer Menge,
- Einfügen (**insert**) von Elementen,
- Entfernen (**remove**) von Elementen,
- Elementetest (**in**),
- Vereinigung (**union**) zweier Mengen,
- Durchschnitt (**intersection**) zweier Mengen,

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768) 2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

- Differenz (**difference**) zweier Mengen sowie die
- Bestimmung der Anzahl der Elemente (**count**) einer Menge.

Eine Sonderrolle spielt der Durchlauf der Elemente mit Befehlsausführung auf jedem Element mit dem **foreach**-Konstrukt, da es sich nicht um eine klassische Operation eines abstrakten Datentyps handelt, sondern um ein höheres Sprachmittel aus funktionalen Programmiersprachen (der Parameter ist eine Funktion, kein Datenwert).

◀Beispiel B-14▶ Die Kombination von Set und Tupel wird oft zur Defintion von komplexen Werten genutzt:

```
Weinkollektion: set(tuple(Weinname: string,
                          Preis: decimal(5,2)))
```

Diese Konstruktion entspricht der Definition einer klassischen Relation. □

Bag-Typkonstruktor

Mengen mit Duplikaten werden als *Multimengen*, engl. *Bag* oder *Multiset*, bezeichnet. Der *Bag-Typkonstruktor* **bag** beschreibt endliche, homogene Multimengen. Im Gegensatz zu **set** sind Duplikate also möglich. Man kann ihn als Verallgemeinerung des Set-Typkonstruktors auffassen und ähnliche Operationen wie für den Set-Typkonstruktor festlegen. Beim Einfügen und Entfernen von Elementen werden Duplikate berücksichtigt, und der Elementetest **in** liefert die Anzahl des Auftretens des angefragten Elements zurück.

Für den Elementedurchlauf sind zwei Versionen möglich: Durchlauf unter Verwendung der Mengen- oder der Multimengensemantik. Die Vereinigung von Bags erfolgt natürlich ohne Duplikateleminierung, und der Durchschnitt ermittelt die minimale Elementeanzahl je Element. Typisches Beispiel für eine Multimenge von Werten sind die Geburtstage einer Gruppe von Menschen: Hier sollten doppelte Einträge nicht verloren gehen.

List-Typkonstruktor

Listen sind ebenfalls homogene Kollektionen, die Duplikate erlauben. Der *List-Typkonstruktor* **list** beschreibt daher endliche geordnete Folgen von Werten. Wie bei Mengen gibt es Operationen zum Erzeugen (**list**) und zum Elementetest (**in**). Listenspezifisch ist das Einfügen an verschiedenen Stellen (**insert**) der Liste und das Aneinanderhängen (**append**) von Listen. Der Elementedurchlauf erfolgt mittels eines Iterators. Der Elementezugriff ist auch über eine Positionsangabe (**[]**) möglich.

◀Beispiel B-15▶ Der List-Typkonstruktor kann zum Beispiel einen Datentyp festlegen, mit dem mehrere Winzer eines Weingutes festgelegt werden:

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768) 2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

```
Winzer: list(tuple(Vorname: string,
                   Nachname: string))
```

□

Array-Typkonstruktor

Arrays sind aus Programmiersprachen wohlbekannt. Der *Array-Typkonstruktor* **array** kombiniert eine feste Anzahl von Werten eines vorgegebenen Datentyps. Es handelt sich also um eine homogene Kollektion. Die Elemente sind über den Index zugreifbar, Duplikate sind erlaubt. Im Gegensatz zu Programmiersprachen erlaubt er nur eindimensionale Arrays, mehrdimensionale Arrays müssen über Schachtelung realisiert werden. Als Operationen definiert er Operatoren zum Erzeugen (**array**) sowie zum Lesen und Schreiben eines Elementes an Indexstelle.

◀Beispiel B-16▶ Als Beispiel für den Array-Typkonstruktor betrachten wir wieder die Möglichkeit mehrerer Telefonnummern. Jedes Weingut kann nun maximal vier Telefone haben, die in fester Reihenfolge angegeben werden:

```
Telefone: array [1..4] of string
```

□

Der Array-Konstruktor spielt in den bekannten Entwurfsmodellen und Objektmodellen keine besondere Rolle, auch wenn er oft aus Programmiersprachen übernommen wurde. Wir werden sehen, dass er in SQL hingegen eine große Rolle spielt: Im Gegensatz zu den bekannteren Konstruktoren wurde er früh in den Standard aufgenommen, wohl aufgrund der leichteren Implementierbarkeit.

Eigenschaften der Typkonstruktoren

Die Eigenschaften der Typkonstruktoren fassen wir in Tabelle B.7 zusammen.

Typ	Duplikate	Elementanzahl	Ordnung	Elementzugriff	Heterogenität
tuple	√	konstant	√	Namen	√
array	√	konstant	√	Index	—
list	√	variabel	√	Iterator/Pos.	—
bag	√	variabel	—	Iterator	—
set	—	variabel	—	Iterator	—

Tabelle B.7: Eigenschaften der Typkonstruktoren

Die homogenen Typkonstruktoren werden auch als Kollektionskonstruktoren bezeichnet. Kollektionen sind homogen, und man kann über die Elemente

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

iterieren (auch über Arrays, indem man in eine Liste umwandelt). Im SQL-Umfeld bedeutet iterieren, dass man im from-Teil eine Tupelvariable an die Kollektion binden kann.

Neben den genannten Operationen gibt es noch Operationen zur Umwandlung von Kollektionen: Arrays können in alle anderen Kollektionstypen, Listen in Multimengen und Mengen, und Multimengen in Mengen umgewandelt werden. Als Operatorname dient jeweils der Name des Zieltyps.

B.4.3.3 Identitäten, Referenzen und Pfadausdrücke

Objekte haben in Objektmodellen eine unveränderliche Identität, über die sie identifizierbar sind, den sogenannten *Objektidentifikator* oder kurz *OID*. Unveränderbar bedeutet hierbei, dass der OID stabil bleibt, auch wenn alle Eigenschaften des Objektes sich ändern sollten. Identifizierbar heißt dass eine OID nicht mehrfach vergeben werden darf. In objektorientierten Programmiersprachen sind OID systemvergeben und nicht interpretierbar.

In RDBMS übernehmen Schlüsselattribute die Rolle der Identifizierung, die allerdings die genannten Eigenschaften nicht erfüllen. Wird beispielsweise der Name des Weinguts zur Identifizierung der Erzeuger genutzt, kann eine Umbenennung aufgrund eines Besitzerwechsels die referentielle Integrität verletzen, da die Einträge dieses Erzeugers in der `WEINE`-Relation nun ins Leere zeigen.

Objektrelationale Modelle nehmen daher Objektidentitäten explizit als neues Konzept hinzu. Im Gegensatz zu Programmiersprachen kennen ORM jedoch mehrere Arten der Generierung von OID, auf die wir bei der Behandlung des SQL:2003-Standards eingehen werden.

Nicht alle Tupel in ORDBMS sind mit OID identifizierbar, nur diejenigen, die zu einem Objekttyp deklariert werden (siehe SQL:2003). Wir reden in diesen Situationen im Folgenden über *Objektrelation* und *Objekttupel*.

Für OID wird ein spezieller Datentyp eingeführt, dessen Werte als *Referenzen* auf Objekttupel genutzt werden können. Durch eine automatische Dereferenzierung sind dann auch *Pfadausdrücke* möglich.

Implementierungstechnisch sind OID mit den Tupelidentifikatoren verwandt, die RDBMS intern zur Adressierung verwenden. Diese sind aber auf der SQL-Ebene nicht nutzbar.

B.4.3.4 Hierarchien und Vererbung

Objektrelationale Datenbankmodelle übernehmen die Idee der Spezialisierung aus Entwurfsmodellen. Im Gegensatz zu objektorientierten Programmiersprachen gibt es also neben der Typhierarchie eine Spezialisierungshierarchie auf den Klassenextensionen. In ORDB sind daher zwei Hierarchien zu betrachten:

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768) 2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

- Die *intensionale* Hierarchie ist eine klassische Typhierarchie auf Objekttypen. Ein Objekttyp ist mit dem Tupelkonstruktor konstruiert. Die Subtypbildung erlaubt die Hinzunahme von Attributen.

- Die *extensionale* Hierarchie ist auf Objekttabellen definiert. Es handelt sich um eine Untermengenbeziehung auf den Instanzenmengen, repräsentiert durch die Mengen der OID der Instanzen. Sie entspricht somit der ISA-Beziehung aus ER-Modellen.

Beide Hierarchien sind gekoppelt: Der Typ t_U einer Untertabelle U zur Tabelle T muss ein (direkter oder indirekter) Untertyp von deren Typ t_T sein.

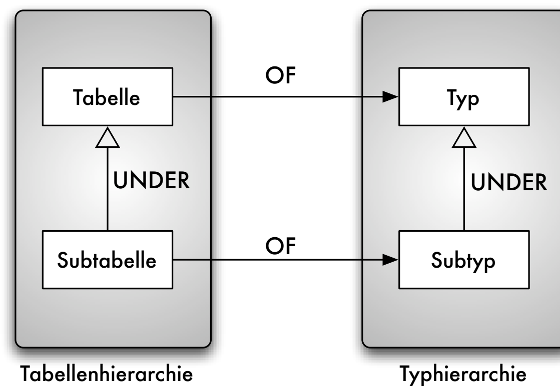

Abbildung B.52: Extensionale und intensionale Hierarchien bei Tabellen

Abbildung B.52 zeigt diesen extensionalen und intensionalen Aspekt von Hierarchien auf Tabellen. Mit **`under`** wird in ORDBMS die Subtyp- bzw. Untertabellenbeziehung bezeichnet, während **`of`** die Typ-Instanz-Beziehung festlegt.

Objekttabellen werden in SQL:2003 als *typisierte Tabellen* bezeichnet und haben folgende Eigenschaften:

- Typisierte Tabellen sind Objekttabellen, deren Typ durch einen „strukturierten Typ“ (Struktur: Tupel mit Attributen) festgelegt wird. Faktisch werden strukturierte Typen durch den Einsatz des Tupelkonstruktors definiert.

- Die Zeilen einer typisierten Tabelle sind über eine *OID-Spalte* referenzierbar.

- Auf den Zeilen einer typisierten Tabelle sind *Methoden* aufrufbar.

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

- Eine typisierte Tabelle kann unter bestimmten Voraussetzungen als *Subtabelle* einer anderen typisierten Tabelle definiert werden.

B.4.3.5 Methoden

Methoden in Objektmodellen entsprechen den gespeicherten Prozeduren aus SQL. In ORDBMS muss somit im Wesentlichen nur Folgendes geklärt werden:

1. Methoden sind gespeicherte Prozeduren (siehe Abschnitt 14.5), die an Objekttypen gebunden sind. Programmiert werden können Methoden in SQL oder extern in einer beliebigen unterstützten Programmiersprache.
2. Methoden werden entlang der Typhierarchie vererbt.
3. Polymorphismus erlaubt das Überladen von Methodennamen und das Überschreiben von Methoden entlang der Vererbungshierarchie.

Die folgenden Arten von Methoden werden unterschieden:

- *Instanzmethoden* entsprechen den klassischen Methoden von Objekten. Sie lesen und ändern potentiell den Objektzustand. Sie werden für eine konkrete Instanz aufgerufen.
- *Statische Methoden* entsprechen den Klassenmethoden der objektorientierten Programmierung. Sie werden unabhängig von konkreten Instanzen aufgerufen.
- *Konstruktormethoden* dienen zur Erzeugung von Instanzen. Aufgerufen werden Sie mit

  ```
  new Typname (Parameterliste ).
  ```

  Eine Konstruktormethode trägt den Namen des zugrundeliegenden strukturierten Typs.

B.4.4 Objektrelationale Konzepte in SQL:2003

SQL:2003 bietet u.a. einen Sammlung von Sprachmitteln zur Unterstützung objektrelationaler Konzepte. Insbesondere wurden nutzerdefinierte Datentypen inklusive Kollektionsdatentypen, Objektidentifikatoren/Referenzen, Vererbung/Spezialisierung und Methoden in den Standard aufgenommen.

Der Standard wird von den aktuellen Systemen bisher nur teilweise umgesetzt: IBM DB2, Oracle und PostgreSQL bieten jeweils eine teilweise Realisierung gemischt mit proprietären objektrelationalen Erweiterungen. Ein umfassendes Buch zu diesem Thema ist das Buch von Türker und Saake [TS05], in dem die Systeme einzeln dem Standard gegenübergestellt werden.

Im Einzelnen bietet der Standard eine Erweiterung um folgende Konzepte:

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768) 2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

- Objektidentitäten und Referenzen sind den objektorientierten Modellen entnommen.
- Ein explizites Typkonzept („Objekttyp“) erweitert das implizite Tupelkonzept von SQL.
- Kollektionstypen werden eingeführt (geschachtelte Relationen).
- Eine Vererbungshierarchie definiert Subtabellen und Subtypen.

Wir werden diese Erweiterungen nun im Einzelnen anhand von Beispielen exemplarisch durchgehen. Eine vollständige Syntaxbeschreibung würde den Rahmen dieses Kapitels sprengen.

B.4.4.1 Typsystem und DDL

Die umfangreichsten Änderungen erfolgten im Typsystem von SQL. Neben der Korrektur einiger Versäumnisse der Vergangenheit (es gibt jetzt einen Datentyp `boolean`) wurde ein Typsystem etabliert, das nutzerdefinierte Typen erlaubt und spezielle Objekttypen kennt.

Typsystem in SQL:2003

Abbildung B.53 gibt einen Überblick über das Typsystem von SQL:2003. Die Tupelkonstruktion kann in SQL durch eine (anonyme) `row`-Konstruktion oder durch Definition eines strukturierten Typs erfolgen.

Abbildung B.53: Typsystem von SQL:2003 im Überblick

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

Die Graphik macht sehr gut deutlich, dass die Kombination eines Objekttypsystems mit dem klassischen relationalen Typsystem zu einem komplexen Gebilde geführt hat, auf das wir im Folgenden im Detail eingehen werden.

SQL:2003: Neue Datentypen

Wir beginnen mit einer kurzen Auflistung der neuen Datentypen und Typkonstruktoren, die in SQL:2003 eingeführt wurden.

- Mit **`blob`** und **`clob`** werden die bereits erläuterten Datentypen für große Binär-/Zeichenkettenobjekte (*Binary / Character Large Objects*) eingeführt.
- Der Typ **`boolean`** beschreibt die booleschen Werte **`true`**, **`false`** sowie (SQL-spezifisch) **`unknown`**.
- Der Typkonstruktor **`array`** wird als Kollektionstyp für mehrwertige Attribute eingesetzt.
- Der Typkonstruktor **`multiset`** ist der Kollektionstyp für ungeordnete Multimengen (mit Duplikaten).
- Die **`ref`**-Typen beschreiben Werte zur eindeutigen Identifikation von Tupeln (mittels OID). Sie werden zur Navigation über Pfadausdrücke genutzt.
- Die **`row`**-Typen entsprechen dem Typkonstruktor **`tuple`** für strukturierte Attribute.
- Nutzerdefinierte Datentypen erlauben die Konstruktion neuer Typen.
- Sequenzen und Identitätsspalten haben wir bereits in Abschnitt 11.4 vorgestellt.

SQL:2003: Kollektionsdatentypen

SQL:2003 unterstützt nur eine Teilmenge der im letzten Abschnitt vorgestellten Kollektionsdatentypen: die Typen **`array`** und **`multiset`**.

◀**Beispiel B-17**▶ Die folgende Tabellendefinition zeigt den Einsatz des **`array`**-Konstruktors (und gleichzeitig auch die Definition eines Large Objects).

```
create table WEINE (
   Name varchar(20),
   Farbe varchar(10),
   Jahrgang int,
   ...
   Foto lob(100K),
   Rebsorten varchar(20) array[7]
);
```


© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

Als Operationen auf **array**-Typen werden Erzeugung und Zugriff angeboten. Die Erzeugung von **array**-Werten über den Arraykonstruktor zeigt die folgende Anweisung:

```
insert into WEINE (Name, Farbe, Jahrgang, Rebsorten)
   values ('La Rose Grand Cru', 'Rot', 2013,
      array['Merlot', 'Cabernet Sauvignon'])
```

Der Zugriff, hier ein lesender Zugriff und ein Überschreiben von **array**-Werten, erfolgt wie in Programmiersprachen gewohnt:

```
update WEINE
set Rebsorten = array['Merlot', 'Cabernet Sauvignon', 'Cabernet Franc']
where Rebsorten[1] = 'Merlot' and Jahrgang = 2013
```

□

SQL:2003 unterstützt den Bag-Konstruktor zur Deklaration von Multimengen. Multimengen als ungeordnete Kollektionen mit Duplikaten wurden mit SQL:2003 eingeführt. Notiert wird die Deklaration des Typs wie folgt:

```
datentyp multiset
```

◄**Beispiel B-18**► Als Beispiel für Multimengen bauen wir unser Beispiel weiter aus.

```
create table WEINE (
   Name varchar(20),
   Farbe varchar(10),
   Jahrgang int,
   ...
   Rebsorten row (Name varchar(20),
                  Anteil decimal(5,2)) multiset)
```

Weine haben nun eine Multimenge von Rebsorten mit den jeweiligen prozentualen Anteilen (mit Duplikaten – den Konstruktor **set** gibt es in SQL:2003 nicht!). Das Beispiel verwendet auch den **row**-Konstruktor zur Tupelkonstruktion.

Die Konstruktion von **multiset**-Werten erfolgt so: Der leere Konstruktor wird als

```
multiset()
```

notiert. Ein Konstruktor mit einer Werteliste wird wie folgt aufgerufen:

```
multiset(row('Merlot', 85.0), row('Cabernet Sauvignon', 12.0),
   row('Cabernet Franc', 3.0))
```


© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

Der Konstruktor kann ebenfalls mit einer Unteranfrage aufgerufen werden:

```
multiset(select Rebsorte as Name, Anteil
         from HERGESTELLT_AUS
         where Weinname = 'La Rose Grand Cru')
```

□

Strukturierte Typen

Nutzerdefinierte Datentypen (kurz UDT für User-defined Data Types) ist der SQL-spezifische Begriff für die Definition abstrakter Datentypen. In SQL:2003 erfolgt eine Unterscheidung in zwei Arten von UDT:

- Die sogenannten *Distinct-Typen* ermöglichen die Definition eines Datentyps direkt als „Umbenennung" eines Basisdatentyps. Dies erhöht die Typsicherheit in Anwendungen, da nun beispielsweise keine Euro-Beträge mehr aus Versehen zu Dollar-Beträgen addiert werden können. Da es zur Mächtigkeit des Datenmodells nicht beiträgt, gehen wir nicht näher auf dieses Konzept ein.

- Die *strukturierten Typen* bieten einen Typkonstruktor für benannte strukturierte Datentypen. Gleichzeitig bilden sie die Grundlage für Objetkttabellen.

Der Typkonstruktor zur Definition abstrakter Datentyp basiert auf der Definition von Tupeltypen und ist verwendbar für Attribute, Parameter, Variablen und Tabellendeklarationen.

◀**Beispiel B-19**▶ Wir definieren einen strukturierten Typ zur Repräsentation von Adressen:

```
create type AdressTyp as (
   Strasse varchar(30),
   Plz char(5),
   Ort varchar(30)
) not final
```

Dieser Datentyp wird nun in unserer ERZEUGER-Tabelle genutzt:

```
create table ERZEUGER (
   Weingut varchar(20),
   ...
   Adresse AdressTyp)
```


© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

□

Die Erzeugung von Instanzen erfolgt durch die Nutzung des Default-Konstruktors `Typname()`. Das folgende SQL-Fragment zeigt die Definition und Nutzung eines überladenen Konstruktors:

```
insert into ERZEUGER (Weingut, Adresse)
   values ('Müller',
   AdressTyp('Am Weinberg 27', '65391', 'Lorch'))
```

Subtyping

SQL:2003 erlaubt die Definition von Untertypen als Erweiterung existierender Datentypen (UDTs). Notiert wird dies wie folgt:

```
untertyp under supertyp
```

Mehrfachvererbung ist nicht zulässig. Bei der Typdefinition wird die weitere Nutzung des Datentyps wie folgt festgelegt:

- Bei der Angabe **`not final`** ist weiteres Subtyping zulässig.
- Die Angabe **`final`** verbietet die Bildung von Untertypen.

Die Untertypbildung kann nun einfach wie folgt erfolgen:

```
create type ErzeugerTyp as ( ... ) not final
create type PrivatErzeugerTyp under ErzeugerTyp as ( ... ) not final
```

Strukturierte Datentypen und die auf ihnen definierte Subtyphierarchie bilden die Grundlage für Objekttabellen und der Spezialisierungshierarchie auf derartigen Tabellen.

Objektidentifikatoren und Referenzen

Werden strukturierte Typen für Tabellen eingesetzt, werden sie implizit zu echten Objekttypen, da die Einträge in diesen Objekttabellen nun einen Objektidentifikator erhalten und referenziert werden können. Die Definition von Tabellen auf Basis strukturierter Typen führt direkt zu einer Objektrelation, in der die Attribute des Typs Spalten werden und eine zusätzliche Spalte für OID erzeugt wird. Diese zusätzliche Spalte wird benötigt für die Unterstützung der **`ref`**-Typen, die wir anschließend behandeln werden.

◀Beispiel B-20▶ Das folgende Beispiel verdeutlicht den Einsatz von strukturierten Typen basierend auf dem Beispiel B-19:

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

```
create typ ErzeugerTyp as (
   Weingut varchar(20),
   Anbaugebiet varchar(20),
   Region varchar(20),
   Adresse AdressTyp
) not final

create table ERZEUGER of ErzeugerTyp (
   ref is oid user generated)
```

- `ErzeugerTyp` ist hierbei ein Tabellentyp, Instanzen können referenziert werden.
- `AdressTyp` ist ein Spaltentyp; Instanzen haben keine Identität und können nicht referenziert werden.

□

Das Beispiel enthält Angaben über die Bildung von Objektidentifikatoren. In objektorientierten Programmiersprachen ist keine derartige Aussage notwendig, da OID dort immer systemgeneriert sind. In SQL:2003 ist die Bildung von Objektidentifikatoren für Tupel in Objektrelationen flexibler. Geblieben ist die Tatsache, dass es um die Definition unveränderlicher Identifikatoren geht. Möglich sind jedoch folgende drei Angaben:

- Mit **`ref is ...user generated`** ermöglicht SQL:2003 die Nutzung nutzerdefinierter und nutzervergebener OIDs.
- Die Angabe **`ref is ...system generated`** legt systemgenerierte OID fest, und entspricht somit der Vorgehensweise in Programmiersprachen.
- Bei der Angabe **`ref from`** (*`Attributliste`*) wird eine OID abgeleitet von anderen Attributen (z.B. vom Primärschlüssel).

Die Nutzung der OID erfolgt über die sogenannten **`ref`**-Typen. Ein Referenztyp ist ein Typ für Referenzen auf Instanzen eines strukturierten Datentyps, also auf Tupel einer Objektrelation. Genutzt werden können Referenztypen für die Repräsentation von 1:1- oder n:1-Beziehungen. Syntaktisch werden sie wie folgt deklariert:

```
ref(strukturierter-typ) [ scope gültigkeitsbereich ]
```

Der Gültigkeitsbereich kann insbesondere eine vorgegebene Tabelle oder alle Tabellen zum betreffenden Typ sein.

◀Beispiel B-21▶ Wir zeigen ein Bespiel, das die Varianten des Einsatzes von **`ref`**-Typen und **`scope`**-Anweisungen zeigt. Der oben definierte Typ `ErzeugerTyp` wird zur Erzeugung der Tabellen für `ERZEUGER` und `HAENDLER` genutzt:

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

```
create table ERZEUGER of ErzeugerTyp ( ... )

create table HAENDLER of ErzeugerTyp ( ... )
```

Schließlich wird die `WEINE`-Tabelle um eine Referenzspalte auf Objekte vom Typ `ErzeugerTyp` erweitert, der Fremdschlüssel kann dadurch entfallen:

```
create table WEINE (
   WeinID int,
   Name varchar(20),
   ...
   Weingut ref(ErzeugerTyp) scope(ERZEUGER)
)
```

Die Angabe des Gültigkeitsbereiches verhindert, dass ein Händler (Objekt aus der Tabelle `HAENDLER`) als erzeugendes Weingut auftritt. □

Als Operation auf Referenztypen wird insbesondere die Belegung von Referenzattributen über Anfragen wie im folgenden SQL-Fragment genutzt.

```
update WEINE
set Weingut = (select oid
               from ERZEUGER
               where Weingut = 'Helena')
where WeinID = 3478
```

Tabellenhierarchien

In SQL:2003 wird die Rolle der Klassen von den Objekttabellen übernommen. Die Subklassen werden daher durch Subtabellen abgebildet. Eine *Subtabelle* ist somit eine Tabelle, die von einer anderen Tabelle abgeleitet ist. Eine Subtabelle erbt die OID-Spalte und alle Spaltenoptionen und kann eigene Spalten hinzufügen. In SQL:2003 gelten die folgenden Einschränkungen:

- Es gibt genau eine direkte Supertabelle, also keine Mehrfachvererbung.
- Der Typ der Subtabelle muss *direkter* Subtyp der Supertabelle sein.

Die syntaktische Notation ist wie folgt:

```
create table subtabellenname of typname
   under supertabellenname [(spalten-optionen)]
```

Die Tabellen- und Typhierarchie wird in Abbildung B.54 noch einmal verdeutlicht.
In SQL:2003 werden *tiefe* und *flache Extensionen* unterschieden:

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

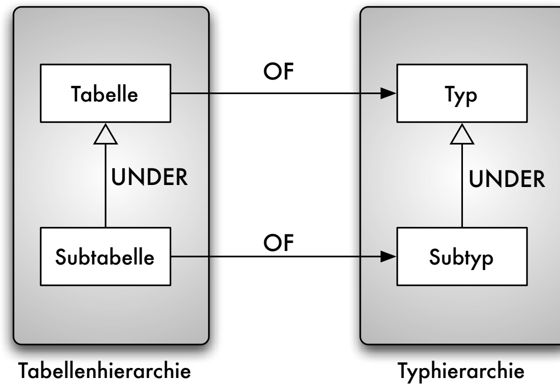

Abbildung B.54: Tabellen und Typhierarchie in SQL:2003

- Die *tiefe Extension* enthält alle Tupel (Objekte) einer Tabelle einschließlich der Tupel der Subtabellen (projiziert auf die originären Attribute der Tabelle).
- Die *flache Extension* ist ohne die Tupel der Subtabellen definiert. Der Zugriff auf flache Extension erfolgt mittels **only**(*tabelle*).

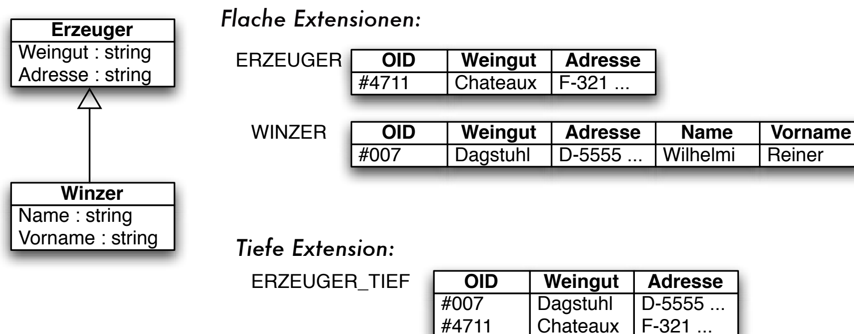

Abbildung B.55: Tiefe und flache Extensionen in SQL:2003

Abbildung B.55 verdeutlicht diese Konzepte anhand eines einfachen Beispiels.

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

B.4.4.2 Anfragen

SQL:2003 realisiert einen Reihe von Erweiterungen der Anfragemöglichkeiten im Zusammenhang mit den besprochenen DDL-Erweiterungen. Aus Platzgründen greifen wir nur einige Erweiterungen exemplarisch heraus.

Operationen auf **array**-*Typen*

Arrays waren die ersten Schachtelungskonstrukte, die in SQL übernommen wurden. Zur Entschachtelung wurde der **unnest**-Operator aus der NF^2-Algebra als neues Sprachkonstrukt übernommen. Die Entschachtelung wandelt eine Kollektion in eine Tabelle um. Wir betrachten hierfür ein einfaches Beispiel.

◀**Beispiel B-22**▶ Wir gehen von einer **array**-Speicherung der Rebsorten eines Weines wie in Beispiel B-17 eingeführt aus, wobei wir uns auf die für dieses Beispiel relevanten Spalten beschränken.

WEINE

Name	Rebsorten
La Rose Crand Cru	**array**[Merlot, Cabernet Sauvignon]
Shiraz	**null**
Zinfandel	**array**[Zinfandel]

Die folgende Anfrage entschachtelt das Array:

```
select *
from WEINE, unnest(Rebsorten) Rebsorte
```

Man beachte, dass die mit **unnest** entschachtelten Tabellen hier implizit mit den Originaltupeln verbunden werden, obwohl die Syntax eher ein Kreuzprodukt erwarten lässt.

Name	Rebsorten	Rebsorte
La Rose Crand Cru	**array**[Merlot, Cabernet Sauvignon]	Merlot
La Rose Crand Cru	**array**[Merlot, Cabernet Sauvignon]	Cabernet Sauvignon
Zinfandel	**array**[Zinfandel]	Zinfandel

Der Eintrag Shiraz, zu dem die verarbeiteten Rebsorten in diesem Beispiel als nicht bekannt angenommen wurden, wird wie bei einem normalen Verbund als Dangling Tuple entfernt. Als Erweiterung kann mittels der Angabe **with ordinality** die ursprüngliche Position als zusätzliche Spalte ausgegeben werden.

```
select *
from WEINE, unnest(Rebsorten) Rebsorte (RName, Pos)
   with ordinality
```

Das Ergebnis mit der Positionsangabe im Attribut Pos lautet dann:

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

Name	Rebsorten	RName	Pos
La Rose Crand Cru	**array**[Merlot, Cabernet Sauvignon]	Merlot	1
La Rose Crand Cru	**array**[Merlot, Cabernet Sauvignon]	Cabernet Sauvignon	2
Zinfandel	**array**[Zinfandel]	Zinfandel	1

□

Weitere Operationen der NF^2-Algebra

Die Entschachtelung lässt sich auch in Anfragen auf Multimengen einsetzen. Der Konstruktor **multiset** erlaubt in SQL:2003 nun auch geschachtelte Kollektionen, wie sie aus dem NF^2-Modell bekannt sind. In Anfragen werden **multiset**-Spalten grundsätzlich als eigene Relationen mittels **unnest** behandelt. In Verbindung mit dem **multiset**-Typkonstruktor lässt sich auf diese Weise auch die orthogonale Schachtelung von Projektion und Selektion der NF^2-Algebra (siehe Abschnitt B.3 im virtuellen Anhang) formulieren. Die vier Grundkombinationen lassen sich durch folgende Anfragemuster [Tür03] darstellen:

Die Kombination *Selektion mit Selektion* wird durch die bereits bekannte Anwendung des **unnest**-Operators ausgedrückt, wodurch auch auf der inneren Tabelle eine Selektion ausgeführt werden kann:

```
select * from ÄußereTabelle
where exists (select *
      from unnest(InnereTabelle)
      where Bedingung)
```

◀Beispiel B-23▶ Für diese und die folgenden Beispielanfragen gehen wir von der erweiterten Definition der WEINE-Relation aus Beispiel B-18 mit den Rebsorten als **multiset** über dem **row**-Typkonstruktor aus. Ein Beispiel einer orthogonal geschachtelten Selektion ist dann die Auswahl aller Weine mit mehr als 20% Merlot-Anteil:

```
select * from WEINE
where exists (select *
      from unnest(Rebsorten) r
      where r.Name = 'Merlot' and r.Anteil > 20)
```

□

Auf die gleiche Weise kann auch die Kombination *Selektion mit Projektion* formuliert werden. Hierbei wird das Prädikat auf der inneren Tabelle mittels **in** nur auf einer Teilmenge der Attribute angewendet:

```
select * from ÄußereTabelle
```


© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

```
where Wertausdruck in (
      select Projektionsliste
      from unnest(InnereTabelle))
```

◀Beispiel B-24▶ Ein konkretes Beispiel ist die Ausgabe der Weine, die u.a. aus Cabernet Sauvignon produziert werden:

```
select * from WEINE
where 'Cabernet Sauvignon' in (
      select r.Name
      from unnest(Rebsorten) r)
```

□

Für die Umsetzung der Kombination *Projektion mit Projektion* kann der **multiset**-Konstruktor mit Unteranfrage nach folgendem Muster eingesetzt werden. In der Unteranfrage ist somit auch die Projektion auf der inneren Tabelle möglich:

```
select multiset(select Projektionsliste
      from unnest(InnereTabelle))
from ÄußereTabelle
```

◀Beispiel B-25▶ Als Beispiel betrachten wir eine Anfrage zur Bestimmung der Rebsorten aller Weine:

```
select Name, multiset(select r.Name
      from unnest(Rebsorten) r) as Rebsorte
from WEINE
```

□

Das gleiche Prinzip kann auch genutzt werden, um die Kombination *Projektion mit Selektion* auszudrücken, indem die Unteranfrage noch um eine Selektion ergänzt wird:

```
select multiset(select Projektionsliste
      from unnest(InnereTabelle)
      where Bedingung)
from ÄußereTabelle
```

◀Beispiel B-26▶ Die Anfrage aus Beispiel B-25 kann hierzu leicht um die Beschränkung auf die Rebsorten mit Anteilen über 10 % erweitert werden:

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

```
select Name, multiset(select *
      from unnest(Rebsorten) r
      where r.Anteil > 10) as Rebsorte
from WEINE
```

□

Nutzerdefinierte Datentypen (UDT)

Wird eine Tupelkonstruktion mittels eines UDT eingesetzt, kann der Zugriff auf Attribute (den Komponenten) auf unterschiedliche Art erfolgen. Die erste Variante nutzt Funktionsaufrufe, bei denen der Komponentenname als Funktion genutzt wird. Notiert wird dies als `attribut(obj)`.

Im Beispiel nutzen wir einen UDT `AdressTyp` mit einer Komponente `Plz`.

```
select * from ERZEUGER
where Plz(Adresse) = '65391'
```

Neben der Funktionsnotation kann auch die vetraute Punktnotation, notiert als `obj.attribut`, genutzt werden.

```
select * from ERZEUGER
where Adresse.Plz = '65391'
```

Methodenaufrufe auf Objekten werden entsprechend Wertausdrücken notiert:

```
Objektwertausdruck.Methodenname [(Parameter)]
```

Für Referenzen wird eine Pfeilnotation zur Dereferenzierung genutzt.

```
Referenzwert->Methodenname [(Parameter)]
```

Natürlich kann man auch mittels eines expliziten **deref** eine Objektinstanz erzeugen, und dann die Punktnotation nutzen.

Man beachte, dass Korrelationsvariablen (Tupelvariablen) einer typisierten Tabelle *nicht* die Instanz repräsentieren! Daher ist folgende Notation unzulässig (basierend auf der Objekttabelle aus Beispiel B-20):

```
select e.berechne_anbaufläche
from ERZEUGER e
```

Korrekt kann diese Anfrage wir folgt formuliert werden:

```
select deref(oid).berechne_anbaufläche
from ERZEUGER
```


© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

Als Operation auf Referenztypen ist die Dereferenzierung (Verfolgen von **ref**-Verweisen) somit in zwei Notationen möglich:

- Der **deref**-Operator liefert alle Attributwerte des referenzierten Objektes.

```
select deref(Weingut)
from WEINE
```

- Der Pfeiloperator erlaubt den Zugriff auf spezielle Komponenten:

```
select Name, Weingut->Anbaugebiet
from WEINE
```

Umgang mit Tabellenhierarchien

Beim Umgang mit Tabellenhierarchien ist die Behandlung tiefer versus flacher Extensionen eine Besonderheit. Wir betrachten wieder unser Erzeuger-Beispiel. Die Definition der Tabellenhierarchie ist wie folgt:

```
create type ErzeugerTyp ( ... ) not final
create type PrivatErzeugerTyp under ErzeugerTyp ( ... ) not final

create table ERZEUGER of ErzeugerTyp
create table PRIVAT_ERZEUGER of PrivatErzeugerTyp
   under Erzeuger
```

Einfügeoperationen auf Subtabellen werden an die Supertabellen propagiert. Eine Einfügung auf PRIVAT_ERZEUGER führt somit zu einer Einfügung auf ERZEUGER. Löschoperationen werden sowohl auf Super- als auch Subtabellen ausgeführt.

Der Zugriff auf Tabellenhierarchien geht per Default von der Behandlung tiefer Extensionen aus. Will man also alle Erzeuger ausgeben, reicht die folgende SQL-Anfrage aus:

```
select * from ERZEUGER
```

Will man alle privaten Erzeuger inklusive der Attribute von ERZEUGER ausgeben, erreicht man dies durch:

```
select * from PRIVAT_ERZEUGER
```

Die Ausgabe flacher Extensionen erfolgt durch die explizite Angabe des Schlüsselwortes **only**. Alle ERZEUGER, die nicht private Erzeuger sind, erhält man somit durch

```
select * from only(ERZEUGER)
```


© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

B.4.4.3 Methoden in SQL:2003

Bei strukturierten Typen kann in SQL die Angabe von Methoden erfolgen. Es handelt sich dabei um normale, in SQL/PSM oder extern implementierte Funktionen/Prozeduren, die UDT zugeordnet sind. Als Besonderheit kann ein impliziter **self**-Parameter (vergleichbar **this** in C++ und Java) genutzt werden. Es erfolgt eine getrennte Spezifikation von Signatur und Implementierung. Zur Vereinfachung gibt es automatisch generierte set/get-Methoden für alle Attribute eines Typs.

◀**Beispiel B-27**▶ Als Beispiel realisieren wir die in Abschnitt 14.5.2 implementierte SQL/PSM-Funktion geschmack als Methoden des Typs WeinTyp:

```
create type WeinTyp as (
   Name varchar(20),
   Jahrgang int,
   Restsuesse int,
   ...
) not final
method geschmack() returns varchar(20))


create method geschmack()
   for WeinTyp
begin
   return case
      when self.Restsuesse <= 9 then 'Trocken'
      when self.Restsuesse > 9 and self.Restsuesse <= 18
         then 'Halbtrocken'
      when self.Restsuesse > 18 and self.Restsuesse <= 45
         then 'Lieblich'
      else 'Süß'
   end
end
```

Der Unterschied zur Implementierung als eigenständiger Funktion besteht im Wesentlichen darin, dass die Methode über **self** direkt auf das Attribut Restsuesse zugreifen kann.

Die Nutzung dieser Methode kann dann u.a. so erfolgen, wenn wir WEINE als Objekttabelle des oben eingeführten Typs annehmen:

```
select Name, Weingut, deref(oid).geschmack()
from WEINE
where Farbe = 'Rot'
```


© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

□

Neben den normalen Instanzmethoden gibt es noch eine Reihe spezieller Methoden wie Konstruktoren, statischen Methoden sowie Funktionen für nutzerdefinierte Typkonvertierung (**cast**-Funktionen) und Ordnungen (**map**- bzw. **ordering**-Funktionen).

Für eine ausführlichere Behandlung von Methoden in SQL:2003 verweisen wir insbesondere auf die Bücher von C. Türker [Tür03, TS05].

B.4.5 Objektrelationale Konzepte in kommerziellen DBMS

Objektrelationale Konzepte werden auch von den kommerziellen Systemen unterstützt, im Fall von Oracle bereits lange vor der Standardisierung in SQL:2003. Leider bestehen dadurch auch einige syntaktische Unterschiede, auf die wir nachfolgend kurz eingehen werden. Wir beschränken uns dabei im Wesentlichen auf Oracle, da sich DB2 weitgehend am Standard orientiert, wobei jedoch keine Kollektionstypkonstruktoren unterstützt werden.

Nutzerdefinierte Datentypen werden in Oracle wie folgt definiert:

```
create type AdressTyp as object ( ... )
```

Auch die Bildung von Subtypen ist möglich, jedoch werden keine Subtabellen unterstützt.

Darauf aufbauend können auch Objekttabellen definiert werden, wobei zur Generierung von Objektidentitäten zwei Möglichkeiten existieren:

- **object identifier is primary key**:

  Nutzung des Primärschlüssels als OID

- **object identifier is system generated**:

  systemgenerierte OID

Für Kollektionen stehen zwei Typkonstruktoren zur Verfügung:

- **varray** ist eine geordnete Multimenge von Elementen, die intern als BLOB gespeichert werden.

- **nested table** ist eine ungeordnete Menge ohne Größenbeschränkung, die intern in einer separaten Tabelle organisiert wird.

◀Beispiel B-28▶ Das Beipiel ist eine Umsetzung von Beispiel B-17 mit den Oracle-Konzepten:

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

```
create type RebsortenListe
   as varray(7) of varchar(20);

create table WEINE (
   ...
   Rebsorten RebsortenListe
);
```

Ein **varray** wird über den Konstruktor erzeugt:

```
insert into WEINE (Name, Farbe, Jahrgang, Rebsorten)
values ('La Rose Grand Cru', 'Rot', 1998,
   RebsortenListe('Merlot', 'Cabernet Sauvignon'));
```

□

Für die Verwendung von geschachtelten Tabellen muss zunächst ein Typ **table of** definiert werden. Außerdem ist eine **store as**-Klausel mit dem Bezeichner für die separate Tabelle notwendig.

◀Beispiel B-29▶ Eine Umsetzung des Beispiels B-18 unter Verwendung von geschachtelten Tabellen könnte in folgender Weise erfolgen:

```
create type RebsortenTyp as object (
   Name varchar(20),
   Anteil numeric));

create type RebsortenTabelle
   as table of RebsortenTyp;

create table WEINE (
   ...,
   Rebsorten RebsortenTabelle)
nested table Rebsorten
   store as weine_rebsorten_tbl;
```

Im Gegensatz zu **varray** können geschachtelte Tabellen auch elementweise geändert werden, allerdings erfordert dies eine Oracle-spezifische Notation:

```
update table (
   select Rebsorten
   from WEINE
   where WeinID = 3478) r
set value(r) = RebsortenTyp(r.Name, 18.0)
where r.Name = 'Merlot';
```


© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

□

Referenztypen werden in Oracle wie im Standard verwendet, wobei jedoch die `oid`-Spalte für die Initialisierung nicht verfügbar ist. Stattdessen muss der `oid`-Wert über die **ref**-Funktion ermittelt werden:

```
update WEINE
set Weingut = (select ref(e)
               from ERZEUGER e
               where Weingut = 'Helena')
where WeinID = 3478;
```

Zum Entnesten muss anstelle der Standard-**unnest**-Operation der **table**-Operator eingesetzt werden:

```
select *
from WEINE, table(Rebsorten);
```

Auch die orthogonale Schachtelung von Projektion und Selektion ist möglich, wobei aufgrund des fehlenden **multiset**-Typkonstruktors hier die **cursor**-Notation notwendig ist:

```
select w.Name, cursor(
      select r.Name, r.Anteil
      from table(w.Rebsorten) r
      where r.Anteil > 20)
from WEINE w;
```

Weiterhin unterstützt Oracle auch Methoden zu Objekttypen, die im eigenen PL/SQL-Dialekt (Abschnitt 14.5.3) implementiert werden.

B.4.6 Zusammenfassung

Objektorientierte Datenbankmodelle entstanden aus der Erkenntnis, dass sich Anwendungsobjekte oft nur umständlich auf Tabellen abbilden lassen. Die Konzepte der Objektdatenbanksysteme sind in die Weiterentwicklung der relationalen Datenbanken hin zu objektrelationalen Datenbanken eingeflossen. Wichtige hinzugenommene Konzepte sind Kollektionsdatentypen zur Unterstützung komplexer Objektstrukturen, ein explizites Typkonzept mit Subtypen, Objektidentitäten, Methoden und Spezialisierungshierarchien. Diese Konzepte werden größtenteils durch die Standards SQL:1999 und SQL:2003 abgedeckt, auch wenn die Umsetzung in den kommerziellen Systemen noch uneinheitlich ist.

Eine Übersicht über die in diesem Kapitel eingeführten Begriffe und deren Bedeutung geben wir in Tabelle B.8.

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

Begriff	Informale Bedeutung
ODBS	Objektdatenbanksystem
objektrelational	Synthese aus Objektorientierung und relationalem Datenbankmodell
Typkonstruktor	parametrisierter Datentyp
Kollektionsdatentypen	Typkonstruktoren für Mengen, Multimengen, Listen und Arrays
UDT	SQL-spezifisch für nutzerdefinierten ADT
flache Extension	Objekte eines Supertyps, die in keinem seiner Subtypen auftauchen
tiefe Extension	Objekte eines Supertyps inklusive der Objekte aller seiner Subtypen

Tabelle B.8: Wichtige Begriffe bei objektrelationalen Systemen

B.4.7 Vertiefende Literatur

Zur Blütezeit der Objektdatenbankforschung wurde intensiv über Objektmodelle publiziert. Damalige Lehrbücher zu objektorientierten Datenbankmodellen, -sprachen und -systemen sind [Heu97, SST97, LV95]. Die ODMG-Norm stellt Cattell in [Cat94, CB97] vor. Relationale und objektorientierte Datenbanken werden in [DLR95] eingeführt, das Schwergewicht der OODBS-Darstellung liegt dabei auf dem System O_2 Weitere Aspekte objektorientierter Informationssysteme werden in [KS96] behandelt.

Im Buch von Türker zu SQL:2003 [Tür03] wird der Standard SQL:2003 umfassend behandelt. Einen allgemeinere Darstellung von objektrelationalen Konzepten findet sich im Buch von Türker und Saake [TS05].

B.4.8 Übungsaufgaben

Übung B-1 Geben Sie ODL-Definitionen an, die die folgenden Entity-Typen und Beziehungen aus unserem Weinbeispiel korrespondieren:

- Kritiker, Gerichte, Weine und Empfehlungen.
- Weine und Schaumweine.
- Kritiken und Weinführer.

□

Übung B-2 Konstruieren Sie eine Spezialisierungshierarchie auf Weinen: Weine, Primeur-Weine, deutsche Weine, Schaumweine und Flaschengärung

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776

als spezielle Schaumweine. Die Klassen haben jeweils eigene Attribute. Geben Sie die unterschiedlichen Abbildungen auf Relationen an. □

Übung B-3 Geben Sie für jeden der Typkonstruktoren ein Beispiel aus dem Wein-Szenario an, wo der Einsatz sinnvoll ist. Versuchen Sie ein sinnvolles Beispiel für jede mögliche zweistufige Schachtelung der Konstruktoren anzugeben (Liste von Listen, Menge von Arrays, etc...). □

Übung B-4 Die Rebsorten mit ihren Prozentanteilen können auch geschachtelt bei Weinen gespeichert werden. Geben Sie eine SQL:2003-Tabelle an, die dies leistet. Geben Sie folgende Anfragen an:

- Alle Weine, die Riesling und Silvaner enthalten.
- Welche Paare von Weinen eines Erzeugers haben dieselbe prozentuale Zusammensetzung?

□

© des Titels »Datenbanken. Konzepte und Sprachen« (ISBN 9783958457768)
2018 by mitp Verlags GmbH & Co. KG, Frechen.
Nähere Informationen unter: http://www.mitp.de/776